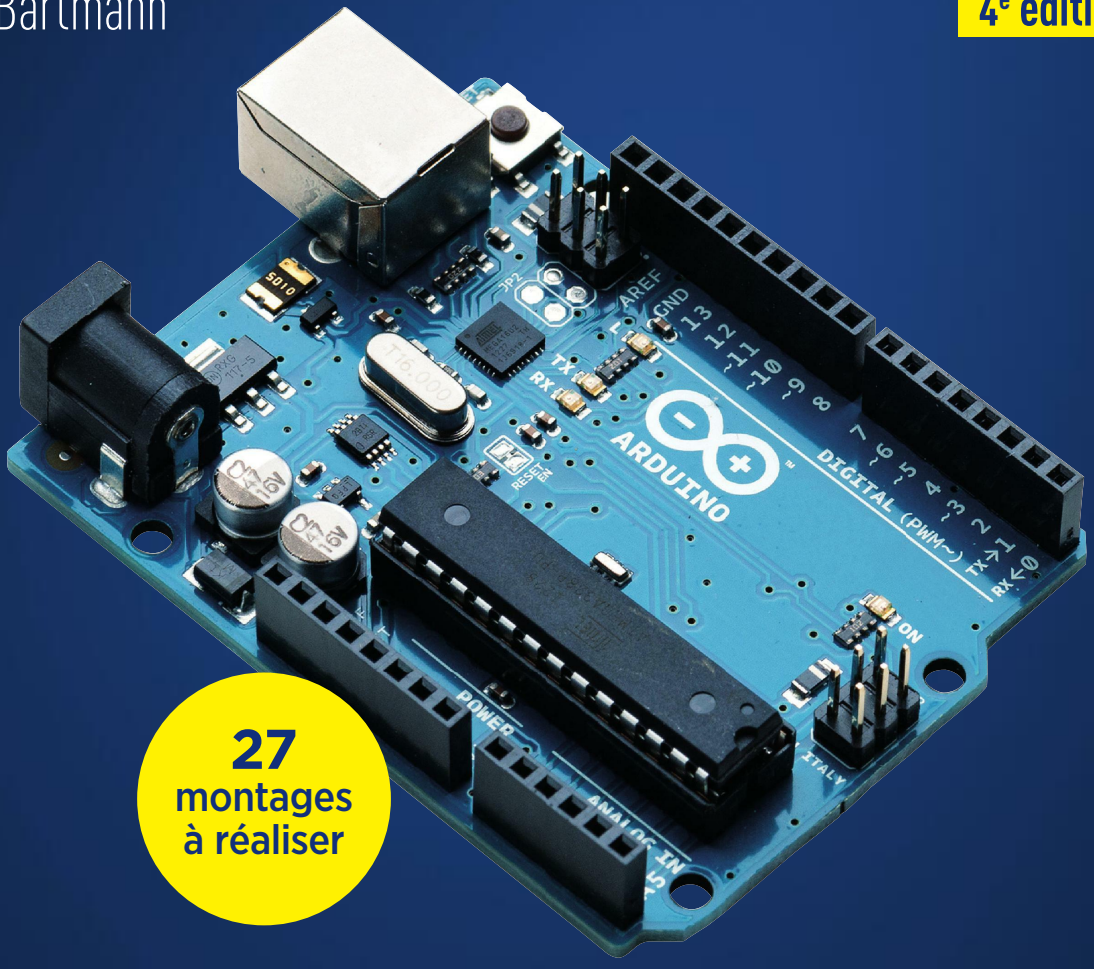
LE GRAND LIVRE D'ARDUINO

Erik Bartmann

4e édition



27 montages à réaliser

Éditions EYROLLES

SERIAL MAKERS

L'ouvrage de référence sur Arduino

Avec son petit microcontrôleur hautement performant et facilement programmable, la carte Arduino a révolutionné le mouvement *Do It Yourself*. Se couplant aisément avec d'autres composants (écrans LCD, capteurs, moteurs...), elle est devenue aujourd'hui un élément indispensable dans de nombreux dispositifs électroniques. Sa simplicité d'utilisation, l'étendue de ses applications et son prix modique ont conquis un large public d'amateurs et de professionnels : passionnés d'électronique, designers, ingénieurs, musiciens...

D'une pédagogie remarquable, cet ouvrage de référence vous fera découvrir le formidable potentiel d'Arduino, en vous délivrant un peu de théorie et surtout beaucoup de pratique avec ses 27 montages à réaliser. Mise à jour avec les dernières évolutions d'Arduino, cette quatrième édition entièrement refondue s'est enrichie de nouveaux projets à monter, qui vous familiariseront notamment avec l'Internet des objets grâce à l'ESP32, Node-RED et MQTT.

À qui s'adresse ce livre ?

Aux makers, électroniciens, bricoleurs, bidouilleurs, ingénieurs, designers, artistes...

Dans ce livre, vous apprendrez notamment à :

- créer un séquenceur de lumière
- fabriquer un afficheur LCD
- commander un moteur pas-à-pas
- faire de la musique avec Arduino

Sur www.editions-eyrolles.com/dl/0100583

Téléchargez le code source des sketches Arduino présentés dans cet ouvrage.

Électronicien de formation, **Erik Bartmann** est aujourd'hui développeur pour le principal fournisseur européen d'infrastructures informatiques. Passionné d'électronique depuis toujours, il est l'auteur de plusieurs ouvrages sur Arduino, Raspberry Pi et l'ESP8266.

SERIAL MAKERS

www.editions-eyrolles.com

LE GRAND LIVRE
D'ARDUINO

CHEZ LE MÊME ÉDITEUR

Dans la collection « Serial Makers »

J. Boyer. – **Réparer son électroménager et ses autres appareils électriques.**
N°0100460, 2022, 432 pages.

R. Sarafan. – **Robots DIY.**
N°0100473, 2021, 192 pages.

D. Nibart. – **46 activités avec le mBot2.**
N°0100628, 2022, 96 pages.

D. Nibart. – **45 activités avec le mBot.**
N°0100270, 2021, 88 pages.

D. Nibart. – **30 défis robotiques.**
N°0100119, 2021, 64 pages.

D. Nibart. – **50 activités avec la carte micro:bit.**
N°0100296, 2021, 80 pages.

J. Boyer. – **Réparez vous-même vos appareils électroniques** (**2e édition**).
N°67621, 2018, 408 pages.

C. Platt. – **L'électronique en pratique** (**2e édition**).
N°14425, 2016, 328 pages.

M. Berchon. – **Le grand livre de l'impression 3D.**
N°67770, 2020, 280 pages.

C. Bosqué, O. Noor et L. Ricard. – **FabLabs, etc.** *Les nouveaux lieux de fabrication numérique.*
N°13938, 2015, 216 pages.

Erik Bartmann

LE GRAND LIVRE D'ARDUINO

4e édition

Éditions EYROLLES

SERIAL MAKERS

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Authorized French translation of the German edition of *Mit Arduino die elektronische Welt entdecken, 4. Auflage* by Erik Bartmann, ISBN 978-3-946496-29-8 © 2021 by Bombini Verlags GmbH. This translation is published and sold by permission of Bombini Verlags GmbH, which owns or controls all rights to publish and sell the same.

Traduction autorisée de l'ouvrage en langue allemande intitulé *Mit Arduino die elektronische Welt entdecken, 4. Auflage* d'Erik Bartmann (ISBN : 978-3-946496-29-8)

Adapté de l'allemand par Catherine Queruel-Haas et Danielle Lafarge

L'éditeur remercie vivement Jean Boyer pour sa validation technique de l'ouvrage.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre français d'exploitation du droit de copie, 20, rue des Grands-Augustins, 75006 Paris.

© Planner/shutterstock pour la photo page 203

© Groupe Eyrolles, 2014, 2015
© Éditions Eyrolles 2018, 2022 pour la présente édition, ISBN : 978-2-416-00583-1

Préface

Cher Erik,

En 2016, je suis tombé sur un de vos livres, qui portait sur l'ESP8266, un microcontrôleur que ma société Espressif avait développé et commercialisé deux ans plus tôt. C'était le premier livre jamais écrit sur l'ESP8266. Bien que je ne parle pas allemand et que, par conséquent, je ne puisse pas le lire, j'ai tout de même compris les affirmations de base que vous avez faites sur l'ESP8266, en parcourant votre texte. Vos belles et nombreuses illustrations m'ont aidé à deviner le texte d'accompagnement. Surtout, j'ai aimé que vous fassiez la promotion du microcontrôleur ESP8266 comme outil du mouvement maker dès 2016. Quoi qu'il en soit, votre évaluation de l'époque s'est avérée correcte : aujourd'hui, dans chaque atelier maker, où que ce soit dans le monde, on peut trouver un ESP8266 ou son successeur, l'ESP32, aux côtés d'un Arduino et d'une carte Raspberry Pi.

Comme je l'ai appris plus tard, vous avez également écrit un livre complet sur Arduino. Depuis sa première publication en 2011, celui-ci a déjà connu trois éditions et s'est imposé comme un ouvrage de référence sur Arduino en allemand et en français. Dans ce livre, vous décrivez non seulement les fonctionnalités du matériel et du logiciel Arduino, mais vous abordez également les thèmes chers aux makers. Vous expliquez les bases de l'électronique de manière très claire, vous donnez une introduction à la programmation et vous montrez dans de nombreux montages tout ce qui peut être développé avec la plate-forme Arduino.

En outre, vous amenez le maker hors des sentiers battus, en lui présentant des projets pointus comme Node-RED ou MQTT qui stimuleront sa créativité. Je suis heureux de voir que vous avez également couvert l'ESP32 avec son propre montage dans votre ouvrage. Je vous souhaite beaucoup de succès pour cette quatrième édition !

Teo Swee Ann, P-DG d'Espressif

Table des matières

Partie I : Les bases

Partie 2 : Les montages

Avant-propos

Arduino pour tous

Lorsque j'ai écrit mon premier livre sur Arduino, il n'existait en Allemagne que deux ou trois distributeurs chez lesquels on pouvait acheter la carte Arduino, tandis que YouTube affichait près de 800 vidéos quand on entrait le mot-clé Arduino. À cette époque, les professionnels de l'électronique ne s'intéressaient guère à ce microcontrôleur et, lorsqu'ils le faisaient, c'était pour se moquer de ses performances.

Les développeurs d'Arduino, qui se sont penchés en 2005 à l'Institut de design d'Ivrée en Italie sur la conception d'un microcontrôleur facile à programmer, ne ciblaient pas à l'origine les professionnels de l'électronique, mais les étudiants en art ayant peu de connaissances en programmation et en matériel informatique. Leur préoccupation d'alors était de pouvoir concrétiser rapidement des idées, de faire du prototypage. L'étudiant pouvait ainsi réaliser lui-même un concept artistique sans l'aide d'ingénieurs ni de programmeurs.

Une erreur de conception qui fait date

Les créateurs de la carte Arduino, qui n'étaient pas des spécialistes en matériel informatique, lancèrent le premier microcontrôleur Arduino, l'Arduino Uno, sous licence libre, transposant ainsi l'expérience du mouvement Open Source au domaine du matériel informatique. Les schémas de connexion étaient en libre accès, de sorte que n'importe qui pouvait compléter ou transformer la carte. Ils mirent également l'environnement de programmation Arduino, également appelé IDE *(Integrated Development Environment)* sous licence Open Source pour que chacun puisse utiliser le logiciel librement.

Bientôt, la carte Arduino ne fut pas utilisée que par des étudiants de l'université italienne, mais s'avéra être une plate-forme de prototypage bien pratique, pour les passionnés d'électronique d'abord, imités ensuite par les amateurs du monde entier. Internet a largement contribué au succès de cette plate-forme en permettant de diffuser facilement les circuits et les logiciels correspondants et en les rendant utilisables par tous grâce à la licence en libre accès. Cela donna naissance à un foisonnement d'idées partout dans le monde, développées par des amateurs fiers de présenter leurs projets à la communauté internationale. Les idées de projet étaient reprises par des amateurs à l'autre bout du monde, améliorées et remises sur le Net.

Durant les années qui suivirent, Arduino et son environnement de développement devinrent quasiment un standard au sein des développeurs amateurs de tous les pays. Une nouvelle plate-forme facile d'accès pour les amateurs d'électronique était née. Les grands fabricants de matériel informatique ne s'y trompèrent pas et remarquèrent que la carte Arduino était de plus en plus utilisée pour toutes les applications possibles de commande et de mesure. Pour élargir les possibilités de la carte Arduino, des platines complémentaires appelées Shields furent développées, qui s'adaptaient exactement à la carte Arduino Uno.

Le jour où l'un des plus grands fabricants de matériel informatique demanda à son département de développement de se baser, à l'avenir, sur la conception physique de la carte Arduino pour son matériel, il était devenu évident que l'Arduino était devenu incontournable dans le monde de la technique. Petite anecdote au passage : les développeurs d'Arduino qui, comme on l'a vu, n'étaient pas à l'origine des spécialistes de matériel informatique, ont construit la carte Arduino Uno avec une erreur de conception. Pour rester compatible avec le quasi standard, le fabricant de renommée mondiale a docilement repris cette erreur de conception, que l'on retrouve encore aujourd'hui sur la carte.

Les bienfaits du copier-coller et ses limites

Je me suis demandé, lorsque j'ai commencé la rédaction de mon premier livre d'Arduino, de quoi un bricoleur avait besoin pour réaliser ses projets avec la carte Arduino. On pouvait déjà à l'époque se procurer des programmes sur Internet, ou via les communautés et forums Arduino naissants, les télécharger sur sa propre carte Arduino et faire fonctionner le projet chez soi, en y ajoutant peut-être quelques interventions ciblées dans le code. Pour installer du matériel supplémentaire, il suffisait de s'orienter sur le circuit pour le reproduire.

Que devrait donc contenir un livre sur Arduino en plus de la description de la carte et de l'environnement de développement et de la manière de transférer les programmes du PC vers le microcontrôleur ? Que se passe-t-il lorsqu'on reproduit un circuit et qu'on y ajoute des composants ? En quête d'une bonne approche conceptuelle pour mon livre, je me suis demandé ce qui m'avait vraiment aidé lors de la réalisation de mes montages électroniques. La réponse est d'une simplicité confondante : mes connaissances de base en électrotechnique m'ont permis de transformer des idées en projets techniques concrets.

Je trouve absolument impressionnant que des personnes qui n'ont rien d'autre à apporter que l'envie de bricoler puissent, avec Arduino, faire fonctionner un circuit de feux de signalisation sur une planche de connexion après seulement une demi-heure, sans avoir jamais tenu un microcontrôleur dans les mains ou écrit une seule ligne de code. Il suffit en effet de regarder une vidéo sur YouTube, et éventuellement une description pas à pas, pour que les LED clignotent au rythme souhaité sur votre table de bureau. On peut alors s'amuser à modifier le rythme en bidouillant dans le code à l'endroit approprié, qui est facile à trouver, et le tour est joué. Mais pour aller plus loin, il faut avoir quelques connaissances de base en électronique et en électrotechnique. C'est pourquoi j'ai écrit ce livre.

Structure de l'ouvrage

Je pense qu'il est important dans un premier lieu de vous présenter le plus de choses possible sur le matériel et le logiciel de la carte Arduino pour que vous fassiez connaissance de toute la palette de fonctions disponibles. Le chapitre 1, « Arduino : le matériel », vous présente de manière assez détaillée la carte Arduino et tous ses composants, l'alimentation électrique et les interfaces. Ce sera également l'occasion de rappeler certains termes de base, usuels dans l'univers de la microélectronique. Chapitre que vous pouvez lire... ou pas. Au chapitre 2, « Arduino : le logiciel », vous apprendrez à contrôler la carte. Vous verrez comment se déroule le flux entre un ordinateur fixe ou portable et la carte Arduino, et comment est construit le code Arduino permettant de commander le microcontrôleur.

Quant au chapitre 3, « N'ayons pas peur de programmer : les bases du codage », celui-ci vous fournira les connaissances de base sur la programmation. Ce chapitre, comme les précédents, va parfois très loin, mais en cas de doute, la règle est la même : vous pouvez le lire, mais vous n'êtes pas obligé de le faire tout de suite. Les premiers chapitres doivent être considérés comme des chapitres de référence, que vous pourrez consulter plus tard au fil de votre lecture ou de la réalisation de vos montages Arduino, si vous voulez avoir des précisions sur un sujet.

Au chapitre 4, « La carte Arduino Discoveryboard », je vous présente une carte de développeur que j'ai développée moi-même. J'ai monté sur une platine des composants électroniques standards qui reviennent en permanence dans les montages Arduino. J'ai soudé ces composants sur la platine, il me suffit ensuite de raccorder la carte Arduino avant de réaliser les projets. L'avantage ? Plus besoin de bricoler sur la plaque de prototypage et la structure du circuit est plus claire. Vous pouvez reproduire la carte Arduino Discoveryboard tout en vous exerçant au soudage.

Il sera ensuite temps de réaliser vos montages Arduino et vos idées de génie ! Les projets sont faciles au début mais ils deviennent rapidement de plus en plus complexes. J'ai ajouté des encadrés de texte aux endroits qui requièrent des connaissances de base pour éclairer le montage d'un point de vue un peu plus théorique. Espérons que vos connaissances en électronique augmenteront au fil des montages et que vous serez capable à votre tour de modifier mes montages ou même de développer les vôtres.

Structure des montages

Comme je l'ai évoqué, les montages vont par ordre de difficulté croissante. Ce que vous avez appris dans un montage vous servira pour les montages suivants. Et si vous avez besoin de connaissances de base, celles-ci vous seront données à l'endroit approprié. Les montages ont tous la même structure :

- analyse du code : je passe en revue le code de programmation étape par étape et j'explique exactement chaque aspect du programme ;
- schéma des connexions : le schéma des connexions est la représentation schématique du montage, le plan de construction pour ainsi dire ;
- structure des connexions : le circuit enfiché sur une plaque de prototypage ou soudé sur une platine est représenté à l'aide de photos ;
- dépannage : la recherche systématique d'erreurs est AMHA (à mon humble avis) l'une des aptitudes particulières que j'aimerais enseigner à tout futur bricoleur électronique. C'est la raison pour laquelle elle fait partie intégrante de mes montages ;
- composants nécessaires : je vous donne un aperçu des composants nécessaires à la réalisation du montage. Vous pourrez ainsi voir tout de suite si vous disposez déjà de ces composants ;
- code de programmation : le code de programmation utilisé pour les montages est exposé, parfois divisé en plusieurs parties.

Pour certains montages, je proposerai des solutions *Quick & Dirty* qui pourront surprendre à première vue. Mais elles seront suivies d'une variante améliorée qui devrait vous faire dire : « Tiens, ça marche aussi comme ça et même pas mal du tout ! Cette solution est encore meilleure ». Si c'est le cas, alors j'aurai atteint le but que je m'étais fixé. Sinon, ce n'est pas grave. Tous les chemins mènent à Rome.

Le code des montages présentés dans cet ouvrage est disponible à l'adresse https://www.editions-eyrolles.com/dl/0100583

Mon site Internet

Sur mon site Internet (en allemand) :

https://erik-bartmann.de/

vous trouverez entre autres des informations supplémentaires sur la carte Arduino. Le meilleur moment pour un auteur est de recevoir les commentaires de ses lecteurs. Je suis parfois vraiment ému lorsqu'un lecteur me dit à quel point il a apprécié la lecture de mon livre et le temps à bricoler. J'en suis très heureux et j'encourage chacun à me dire ce qu'il a pensé de mon livre, ce qu'il a particulièrement apprécié et aussi ce qu'il estime devoir être amélioré. Écrivez-moi, mon adresse e-mail est :

erik.bartmann@yahoo.de

Prérequis

Le seul prérequis personnel est d'aimer le bricolage et les expériences. Nul besoin d'être un geek ni un expert en ordinateur pour comprendre et reproduire les montages de ce livre. Nous commencerons en douceur afin que chacun puisse suivre. Ne vous mettez pas de pression, le premier objectif de cet ouvrage est de vous distraire !

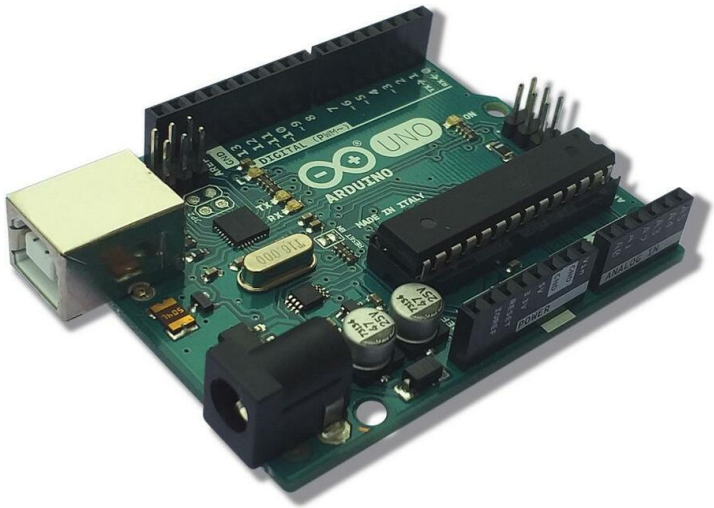
Composants nécessaires

Notre carte Arduino est certes bien sympathique, et nous apprécions que tout y soit si bien pensé et si petit. Mais nous verrons au cours d'une prochaine étape tout ce que nous pouvons y raccorder de l'extérieur. Si vous n'avez pas l'habitude de manipuler des composants électroniques (résistances, condensateurs, transistors, diodes...), pas d'inquiétude. Chacun fera l'objet d'une description détaillée, afin que vous sachiez comment il réagit individuellement et au sein du circuit. Au début de chaque montage,

j'indiquerai, comme je l'ai dit plus haut, la liste des composants nécessaires que vous devrez vous procurer. Naturellement, l'élément-clé de tout circuit sera toujours la carte Arduino, mais je ne la mentionnerai pas forcément de manière explicite.

À ceux qui se demandent combien coûte une carte Arduino et s'ils peuvent conserver leur train de vie après un tel achat, je répondrai : « *Yes, you can !* ». Elle est très bon marché, aux alentours de 25 euros. J'utilise la carte Arduino Uno dans tous les chapitres importants. Les prix des autres composants que nous utiliserons dans les montages sont également très abordables.

Figure 1 ▶
La carte Arduino Uno – l'originale



Règles de conduite

Lorsque vous êtes dans le feu de l'action et que vous vous concentrez sur quelque chose qui vous passionne, les effets suivants se produisent :

- diminution de la consommation de nourriture, ce qui peut entraîner une perte de poids critique et une perte inquiétante de la réalité ;
- une hydratation insuffisante pouvant aller jusqu'à la déshydratation et l'augmentation de la formation de poussières dans l'environnement ;
- abandon de toutes les mesures d'hygiène comme se laver, prendre une douche, se brosser les dents, associé à une présence accrue de vermine ;
- rupture de toutes les relations sociales.

N’attendez pas d’en arriver là et ouvrez de temps en temps la fenêtre pour permettre aux insectes de quitter la pièce et laisser entrer l’air frais et la lumière du soleil. Pour contrer les effets mentionnés précédemment, vous pouvez régler votre réveil afin d’être invité à interrompre votre activité à intervalles réguliers. Après la publication de ce livre, je ne veux pas être confronté à une vague de plaintes de la part de partenaires en colère ou d’amis qui ont été négligés. Vous ne pourrez pas me dire que je ne vous aurai pas prévenu. Il ne me reste plus qu’à vous souhaiter un bon divertissement et une bonne réussite avec votre carte Arduino !

Partie I

Les bases

Chapitre 1

Arduino : le matériel

Comme je l'ai déjà évoqué dans l'avant-propos, le projet Arduino a été développé à l'Interactive Design Institute d'Ivrée en Italie. C'est dans un bar situé près de ce lieu que se retrouvaient régulièrement Massimo Banzi et David Cuartielles, qui développèrent la première carte Arduino en 2005. Le bar portait le nom d'Arduin d'Ivrée, qui fut roi d'Italie autour de l'an 1000. *Arduino* désigne aujourd'hui la plate-forme logicielle et matérielle de ce projet Open Source.

Le présent chapitre traite de la plate-forme matérielle Arduino. J'y explique les termes de base et les sujets indispensables pour connaître la technique du microcontrôleur. Je passe également en revue les composants qui constituent un microcontrôleur. Comme ces composants se retrouvent pratiquement dans n'importe quel microcontrôleur, il est important de les présenter en détail. Traiter des thèmes de base comme la tension ou le courant me donnera aussi l'occasion de rafraîchir un peu vos connaissances en physique.

Les différentes cartes Arduino

Nous travaillerons dans ce chapitre, et dans tout le livre, avec la carte Arduino Uno. Elle a été la première carte à être développée et produite par les fondateurs d'Arduino. Par la suite, d'autres cartes Arduino avec des améliorations techniques sont apparues. Quel genre d'améliorations ? Si l'on choisit des caractéristiques telles que la fréquence du processeur ou la mémoire vive disponible comme critères de décision pour l'achat d'une nouvelle carte Arduino, il y a certainement des cartes qui conviennent mieux parce qu'elles fonctionnent plus rapidement et qu'elles peuvent stocker et traiter des programmes plus volumineux. Mais ce n'est pas forcément synonyme d'amélioration. Pour faire vos premiers pas, la carte Arduino Uno est à mes yeux la meilleure solution, parce qu'elle est très solide et très répandue.

La carte Arduino Yún par exemple est une carte intéressante ; elle offre en effet un certain nombre d'extensions, comme le système d'exploitation Linux. Toutefois, cette carte ne s'est pas vraiment imposée au sein des

développeurs amateurs, peut-être parce que le Raspberry Pi était déjà sur le marché lorsque Arduino Yún a été lancée. Être plus rapide ou avoir plus de fonctionnalités ne signifie pas obligatoirement être meilleur.

J'aimerais pourtant vous indiquer quelques cartes Arduino notables :

- Arduino Leonardo ;
- Arduino Mega 2560 ;
- Arduino Nano.

Ces cartes se distinguent par leur taille et le nombre de prises, donc de possibilités de connexion pour communiquer avec le monde extérieur. Elles présentent en outre des caractéristiques différentes pour ce qui est du processeur, de la fréquence de cadence et du volume de mémoire. Néanmoins, elles fonctionnent toutes suivant le même principe, et peuvent être adressées et programmées avec un seul et même environnement de développement Arduino. Selon le domaine d'application et les exigences requises, une carte Arduino est peut-être plus appropriée qu'une autre. Certains auront besoin d'une carte avec de nombreuses broches E/S et opteront par exemple pour la carte Arduino Mega ou la Due. D'autres choisiront la carte Arduino Micro ou Nano, car elles sont toutes petites et rentrent très bien dans des boîtiers petit format. Elles sont utilisées pour des applications où l'espace est restreint.

Le génie toute catégorie est selon moi la carte Arduino Uno et elle le restera pendant encore longtemps. Elle offre une plate-forme idéale pour ceux qui débutent dans l'univers des microcontrôleurs. La plupart des tutoriels, projets et discussions sont également accessibles sur Internet. Lorsque vos montages deviendront de plus en plus ardus, vous n'aurez aucun problème à acheter un autre modèle de carte Arduino, car les prix sont vraiment abordables. De nombreux amateurs se procurent plusieurs cartes différentes au fur et à mesure pour compiler les expériences, ce qui est à mes yeux tout à fait normal.

Les liens ci-dessous vous permettent d'accéder à des informations détaillées sur les cartes que j'ai mentionnées :

- Arduino Uno :
  https://www.arduino.cc/en/Main/ArduinoBoardUno
- Arduino Mega :
  https://www.arduino.cc/en/Main/ArduinoBoardMega2560
- Arduino Leonardo :
  https://www.arduino.cc/en/Main/ArduinoBoardLeonardo
- Arduino Micro :
  https://www.arduino.cc/en/Main/ArduinoBoardMicro

- Arduino Nano :
  https://www.arduino.cc/en/Main/ArduinoBoardNano

Il existe de nombreuses autres cartes Arduino et d'extensions, que vous trouverez aux adresses suivantes :

- https://www.arduino.cc/en/Main/Products
- https://www.arduino.cc/en/Main/Boards

La carte Arduino Uno

Commençons par examiner de plus près la carte Arduino. J'ai identifié sur la figure ci-dessous les principaux composants de la platine.

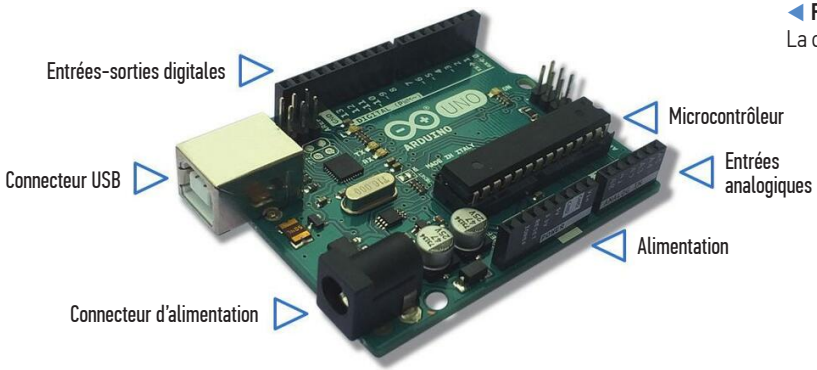


Figure 1
La carte Arduino Uno

Nous allons passer en revue ces différents éléments. Tout ne vous paraîtra peut-être pas encore très clair, mais je vous promets que nous les examinerons en détail un peu plus loin.

Le microcontrôleur

Le *microcontrôleur* est le cœur de la carte Arduino. Il est quasiment le centre de calcul de la carte Arduino. La carte Arduino Uno que nous utiliserons est équipée d'un microcontrôleur Atmel AVR de type *ATmega328*. Sur la figure 1 ci-dessus, c'est le gros composant noir muni de nombreuses broches. Ce composant est fabriqué par la société américaine Microchip Technology Inc. et se retrouve dans de nombreuses cartes.

Voici, page suivante, une petite vue d'ensemble des principales caractéristiques techniques de la carte Arduino Uno.

Tableau 1 ▶
Quelques caractéristiques intéressantes de la carte Arduino Uno

Catégorie	Valeur
Microcontrôleur	ATmega328
Tension de service	5 V
Tension d'entrée (recommandée)	7 V à 12 V
Tension d'entrée (limites)	6 V à 20 V
Broches E/S numériques	114 (6 sorties commutables en MLI)
Entrées analogiques	6
Courant CC par broche d'E/S	40 mA
Courant CC pour broche 3,3 V	50 mA
Mémoire Flash	32 Ko (ATmega328), dont 0,5 Ko est utilisé par le chargeur d'amorçage
SRAM	2 Ko (ATmega328)
EEPROM	1 Ko (ATmega328)
Fréquence d'horloge	16 MHz

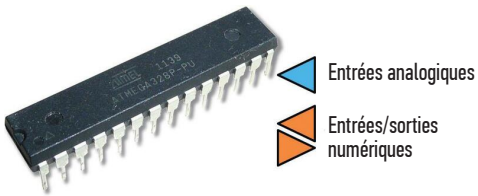
Vous trouverez des informations détaillées à l'adresse Internet suivante :

https://www.arduino.cc/en/Main/ArduinoBoardUno

Les entrées et sorties

Pour communiquer avec un microcontrôleur, il nous faut établir une connexion physique, quelle que soit sa forme. Ces entrées et sorties sur la carte Arduino sont appelées des ports. Comme vous pouvez le voir sur le tableau 1, nous disposons d'un certain nombre d'entrées et de sorties pour communiquer avec la carte Arduino. Elles constituent l'interface avec le monde extérieur et permettent d'échanger des données avec le microcontrôleur. Jetons un œil sur la figure ci-dessous :

Figure 2 ▶
Les entrées et sorties numériques de la carte Arduino Uno



Vous y voyez le microcontrôleur qui peut communiquer avec nous par le biais de certaines interfaces. Certains ports servent d'entrées, d'autres d'entrées et de sorties.

QU'EST-CE QU'UN PORT ?

Un port est un chemin d'accès défini vers le microcontrôleur, une sorte de porte qui mène à l'intérieur et qui permet de communiquer avec ce dernier. Les broches d'entrée et de sortie sont généralement organisées en ports et affectées dans un microcontrôleur à un *registre*, c'est-à-dire une certaine plage de mémoire.

Regardez la carte, vous apercevez des réglettes de raccordement noires sur ses bords supérieur et inférieur. Vous vous demandez sûrement par exemple quelles entrées et sorties numériques sont utilisées pour les entrées et les sorties ? Peut-être que certaines ne fonctionnent que comme des entrées et d'autres que comme des sorties. Cela manquerait de flexibilité. C'est pour cela qu'il existe une bien meilleure solution. Et qu'en est-il des sorties analogiques ? Il n'y en a pas du tout ici.

Commençons par les broches numériques. Chaque broche est le raccordement séparé d'un port, qui combine plusieurs broches. Comme les spécifications l'indiquent, la carte Arduino Uno dispose de 14 broches numériques d'entrées et de sorties, ou broches E/S, E/S étant l'abréviation d'entrée et de sortie. On peut alors programmer de manière très flexible quelle broche parmi les 14 fonctionnera comme entrée ou comme sortie. Ce processus est appelé *configuration*.

Passons maintenant aux broches analogiques. Notre carte Arduino ne dispose pas de sorties analogiques séparées. Cela peut paraître bizarre au premier abord, mais certaines broches numériques sont détournées de leur destination première et servent de sorties analogiques. Vous vous demandez certainement comment cela fonctionne. Voici donc un avant-goût de ce qui sera expliqué dans le montage n° 1 sur la *modulation de largeur d'impulsion*, ou *MLI*. Il s'agit d'un procédé dans lequel le signal présente des phases à niveau haut et des phases à niveau bas plus ou moins longues. Si la phase à niveau haut, dans laquelle le courant circule, est plus longue que celle à niveau bas, une lampe branchée par exemple sur la broche correspondante éclairera visiblement plus fort que si la phase à niveau bas était la plus longue. Plus d'énergie sera donc apportée en un temps donné sous forme de courant électrique À cause de la persistance rétinienne de notre œil, nous ne pouvons différencier des événements changeant rapidement que sous certaines conditions, et un certain retard se produit aussi lorsque la lampe passe de l'état allumé à celui éteint, et réciproquement. Cela m'a tout l'air d'être une tension de sortie qui se modifie, bizarre, non ? Vous comprendrez certainement mieux lorsque nous aborderons ce chapitre.

En tout cas, ce mode de gestion des ports présente d'emblée un inconvénient. Quand vous utilisez une ou plusieurs sorties analogiques, c'est au

détriment de la disponibilité des ports numériques, il y en a alors d'autant moins à disposition, mais cela ne saurait nous gêner outre mesure, car nous n'atteignons pratiquement pas les limites de la carte. De ce fait, nous n'avons pas de restriction sur les montages expérimentaux à tester. Le terme *chargeur d'amorçage* (ou *bootloader* en anglais) apparaît dans les spécifications de la carte et nécessite une explication.

QU'EST-CE QU'UN CHARGEUR D'AMORÇAGE ?

Un chargeur d'amorçage est un petit programme qui doit être installé une fois sur le microcontrôleur ATmega328. Déjà installé par le fabricant lorsque vous achetez une carte Arduino, ce programme a sa place dans une certaine zone de la mémoire flash du microcontrôleur et assure le chargement du logiciel une fois la mise sous tension établie. Celui-ci s'exécute alors automatiquement, sans attendre qu'il soit enregistré dans la mémoire.

La mémoire flash disponible est diminuée de la part attribuée au chargeur d'amorçage. Normalement, un microcontrôleur reçoit son programme de travail d'un matériel informatique supplémentaire, par exemple d'un programmateur *In System Programming* (ISP). Le chargeur d'amorçage évite cela, ce qui rend le téléchargement du logiciel vraiment facile. Sitôt dans la mémoire de travail du contrôleur, le programme de travail est exécuté. Si jamais vous deviez changer, pour une raison quelconque, votre microcontrôleur ATmega328 sur la carte, le nouveau circuit ne saurait pas ce qu'il doit faire, car le chargeur d'amorçage n'est pas enregistré par défaut. Cette fonctionnalité peut être installée selon différentes procédures que je ne peux pas expliquer ici faute de place. Cependant, vous trouverez sur Internet suffisamment d'informations pour vous permettre d'installer le chargeur d'amorçage approprié au microcontrôleur.

L'alimentation électrique

Commençons par la tension d'alimentation, car sans elle rien n'est possible. Il existe différentes possibilités. Quand nous travaillons avec Arduino ou que nous le programmons, il est indispensable d'établir une connexion USB avec l'ordinateur. Cette liaison assure deux fonctions :

- transmettre l'indispensable tension d'alimentation de 5 V ;
- offrir un canal de communication entre l'ordinateur et la carte Arduino.

Les deux fonctions sont remplies par le port USB argenté, désigné sous le nom de *connecteur USB* sur la figure 1. Dans le chapitre suivant, nous examinerons la prise de plus près, ainsi que son type et la procédure d'installation du logiciel requis. Ce chemin d'accès à la carte est employé pour le développement et le test de programme. Une seconde prise se trouve juste à côté de la première : c'est la prise d'alimentation, appelée *connecteur*

d'alimentation. Elle permet de déconnecter la carte de l'ordinateur une fois sa programmation effectuée. À quoi cela sert-il ? C'est très simple ! Imaginons que nous construisions un robot motorisé et que la carte Arduino doive réceptionner des commandes transmises par un émetteur radio et exécuter les actions correspondantes. Le câble USB limiterait considérablement le rayon d'action du véhicule. Nous avons simplement besoin d'une alimentation électrique pour la carte et éventuellement pour les moteurs, car la programmation est terminée. Comme le déclare la NASA au décollage des fusée : *le guidage est interne*. La carte Arduino est autonome ! La figure 3 ci-dessous présente une alimentation électrique externe à l'aide d'une pile 9 V, ce qui toutefois ne constitue pas une solution durable.

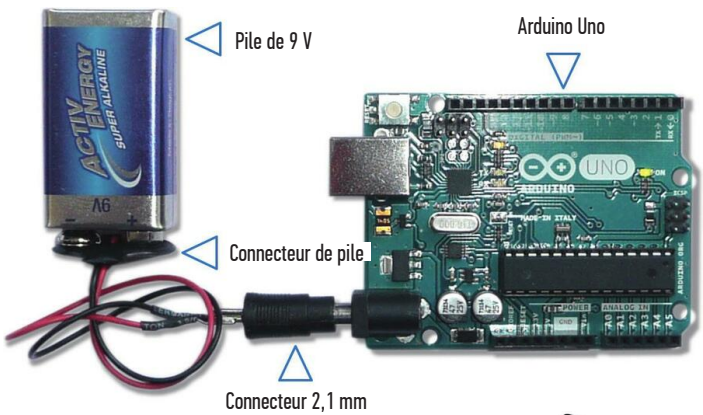


Figure 3
La carte Arduino Uno avec une alimentation électrique externe via une pile

La tension d'alimentation doit être comprise entre 7 et 12 V CC (CC = courant continu). Un bloc d'alimentation à fiche avec une prise de 2,1 mm à moins de 10 € constitue la meilleure variante d'alimentation électrique externe. Il doit par exemple fournir une tension de 9 V CC/1 A :

Figure 4
Une alimentation électrique avec bloc d'alimentation à fiche

Il existe aussi une broche *Vin*, qui peut servir également à l'alimentation électrique. Vous trouverez des informations détaillées aux adresses Internet suivantes :

https://www.arduino.cc/en/Main/ArduinoBoardUno

https://playground.arduino.cc/Learning/WhatAdapter

COMPRENDRE LES TERMES DE BASE

À propos de l'alimentation électrique

Savez-vous ce qu'est la tension et comment elle est produite ? Nous n'avons pas non plus abordé la question du courant. Vous avez l'occasion de rafraîchir ici vos connaissances de base en électronique et de découvrir davantage de possibilités avec votre carte Arduino.

Grandeurs physiques

Étant donné que sans la présence de tension, les appareils électriques ne sont rien d'autre que des appareils inutiles, j'aimerais aborder brièvement les grandeurs physiques. Les grandeurs physiques ou grandeurs fondamentales électriques sont principalement la tension électrique, le courant électrique, la résistance électrique et la charge électrique. Pour pouvoir construire vos propres projets avec la carte Arduino et ne pas vous contenter de les reproduire, il est indispensable de comprendre le lien entre tension, courant et résistance. Cet ouvrage n'étant pas conçu comme un manuel d'électronique, mais plutôt comme un guide pratique sur Arduino, je n'aborderai le sujet que de manière succincte.

Qu'est-ce que la tension ?

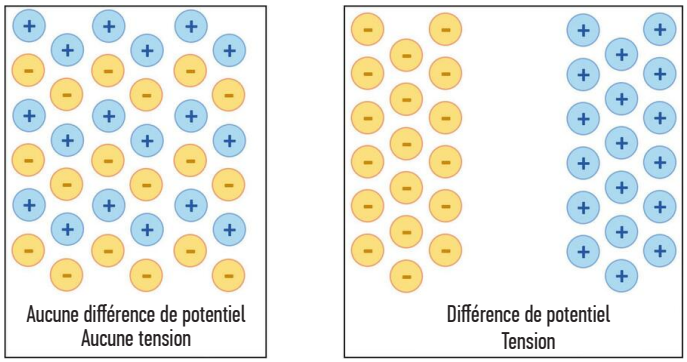
Tous les matériaux qui nous entourent, qu'ils soient solides, liquides ou gazeux, sont composés d'atomes. Un atome est composé d'un noyau et d'une enveloppe, le noyau étant constitué de protons chargés positivement et de neutrons. Dans l'enveloppe de l'atome, les électrons chargés négativement évoluent autour du noyau. Si le nombre d'électrons est réparti uniformément, nous ne remarquons en fait rien d'un phénomène électrique, sauf si l'équilibre électrique est perturbé d'une manière ou d'une autre. Si, par exemple, ces porteurs de charge négative, représentés par les électrons, sont retirés d'un corps et ajoutés à un autre corps, il existe alors un état électrique entre les deux corps qui a rompu l'équilibre de charge qui existait précédemment. Plus ce déséquilibre est important, plus l'état électrique est important.

On pourrait comparer cela à deux personnes qui ne seraient pas du même avis et entre lesquelles se formerait alors une certaine tension. Plus les divergences sont importantes, plus la tension est grande. Et c'est ainsi que le premier aspect d'un état électrique a été désigné par le terme de tension électrique.

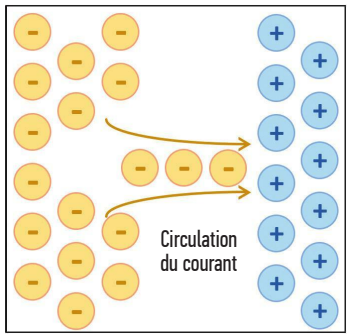
La séparation des charges est un état de repos, donc statique, car rien ne bouge après la séparation des porteurs de charge. La tension en présence peut être mesurée. L'unité de mesure de la tension est le volt, dont l'abréviation est la lettre V.

Jusqu'ici tout est clair, mais à quoi sert cette différence de charge, qui représente une certaine tension ? L'état statique change brusquement lorsqu'une liaison plus ou moins conductrice est établie entre les deux corps, liaison que les électrons peuvent enjamber. Cela peut se produire avec les matériaux les plus divers, comme le cuivre, l'aluminium ou l'argent.

La figure 5 montre sur la gauche une répartition uniforme de porteurs de charge négatifs et positifs. Les deux sont équilibrés et il n'y a donc pas de différence de potentiel et pas de tension. En revanche, sur la droite, les porteurs de charge sont séparés. La partie gauche est dominée par les porteurs de charge négatifs, la partie droite par les porteurs de charge positifs. Cette différence de potentiel crée une tension.



Un déséquilibre de charge établi tend toujours à s'équilibrer et les électrons se déplacent d'un corps à l'autre lors d'une liaison, la migration se faisant du surplus d'électrons vers le manque d'électrons. C'est seulement lorsque cette possibilité est donnée qu'un flux de courant circule, comme on le voit sur la figure 6 :



◀ **Figure 6**
Circulation d'un flux de courant

Vous savez déjà qu'il existe différentes formes de tension, appelées *tension continue* et *tension alternative*. La figure 7 page suivante montre ces formes de tension au cours du temps, la carte Arduino fonctionnant avec une tension continue (ou courant continu). Cette forme de tension, on parle aussi de courant continu, est caractérisée par une intensité et une direction qui ne varient pas. Elle est symbolisée par les lettres *DC* (*Direct Current*) ou, en France, par CC (courant continu). Le courant alternatif est, quant à lui, noté *AC* (*Alternating Current*) ou CA (courant alternatif). D'où le nom du groupe de rock AC/DC. Cette forme de courant est régalement représentée sur la figure 7.

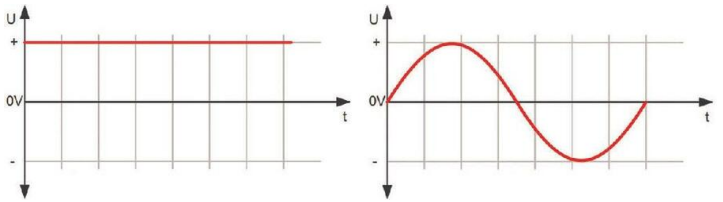


Figure 7 ▶
Circulation d'un flux de courant

La hauteur de la tension U du courant continu n'évolue pas, tandis que la valeur de la tension U du courant alternatif varie en permanence et oscille entre deux valeurs limites, l'une positive et l'autre négative, en formant une courbe sinusoïdale.

Qu'est-ce que le courant ?

Vous savez maintenant qu'un courant électrique peut passer en présence de tension. Mais de quoi s'agit-il ? Que se passe-t-il dans un matériau conducteur comme le cuivre lorsqu'il est traversé par du courant ? La présence d'une tension et la différence de potentiel qui en résulte font que les électrons présents sur l'enveloppe la plus externe des atomes sont arrachés et se déplacent sous forme d'électrons libres comme un nuage à travers le matériau. Cela s'explique par le fait qu'en libérant les électrons libres, un atome devient ce que l'on appelle un *ion*. Un ion est un atome présentant un déséquilibre entre les électrons et les protons. Vers l'extérieur, l'atome n'est plus neutre. Un électron arraché laisse derrière lui un vide dans la composition de l'atome, qui peut être comblé par un autre électron libre.

Ce processus d'arrachement et de remplissage des vides par les électrons est désigné par le terme de courant électrique. Les deux figures ci-après représentent le flux de courant à travers un conducteur. La section transversale du conducteur et la période d'observation pour mesurer le courant jouent un rôle décisif. On voit sur la première figure six électrons entiers par segment indiqué. Ce n'est pas beaucoup, comme on le verra plus tard, mais pour le moment, cela ne joue qu'un rôle secondaire.

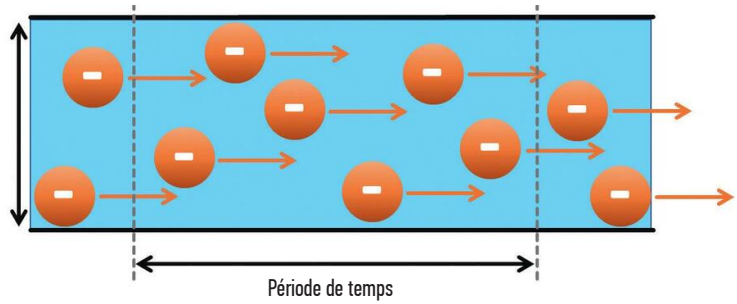
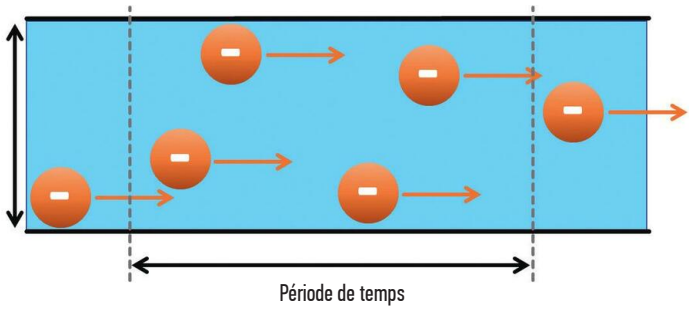


Figure 8 ▶
De nombreux électrons circulant à travers un conducteur

Sur la deuxième figure, on voit comparativement moins d'électrons, seulement quatre par segment indiqué. Le flux de courant est donc plus faible.



◀ **Figure 9**
Quelques électrons circulant à travers un conducteur

Le flux électrique peut donc être défini avec le rapport suivant :

$$Puissance\ du\ courant = \frac{Nombre\ d'électrons}{Période\ de\ temps}$$

Bien entendu, le diamètre du conducteur joue également un rôle, car plus le canal de passage des électrons libres est étroit, plus la résistance est grande et plus le flux de courant est faible. Comme les électrons possèdent une charge, la somme des électrons à considérer par période de temps peut aussi être considérée comme une *quantité de charge*. Cette quantité de charge est symbolisée et désignée par la lettre *Q*. On peut maintenant exprimer le tout de façon plus précise par une formule mathématique :

$$I = \frac{\Delta Q}{\Delta t}$$

La lettre *I* symbolise le courant et le petit triangle Δ, appelé delta, est utilisé en mathématiques pour symboliser les changements ou les différences. Cela signifie donc que l'intensité du courant est égale à la modification de la quantité de charge *Q* par le temps *t*. Dans l'exemple ci-dessus avec six ou quatre électrons par intervalle de temps, on peut constater que c'est très, très, très peu et le calcul suivant se rapproche un peu plus de la réalité. On souhaite déterminer le nombre d'électrons qui traversent le conducteur pendant un intervalle de temps d'une seconde sous une intensité de 1 A (A signifie ampère, soit l'unité de mesure du courant électrique). Mais comment calculer cela ? Pour ce faire, on inverse la formule indiquée selon la quantité de charge, qui se présente alors comme suit :

$$\Delta Q = I \cdot \Delta t$$

L'équation se présente comme suit avec les valeurs saisies :

$$\Delta Q = 1\text{A.s}$$

Comme on l'a vu, la quantité de charge *Q* est le nombre *n* d'électrons et il est bon de savoir quelle est la charge élémentaire d'un électron pour résoudre le problème posé. Celle-ci est symbolisée par la lettre *e*, qui a comme valeur $1{,}602176 \cdot 10^{-19}$ C, où C signifie coulomb, l'unité de mesure d'une charge électrique. On peut alors remplacer dans la dernière formule ΔQ par le nombre de charges élémentaires *n* et on obtient :

$$n \cdot e = 1\text{A.s}$$

On voit très bien ici que 1C (coulomb) peut aussi présenter l'unité A.s, ce qui apparaît immédiatement dans la formule abrégée. On obtient :

$$n = \frac{1\text{A.s}}{1,602176 \cdot 10^{-19}\ \text{A.s}} = 6,24151 \cdot 10^{18}$$

Le résultat impressionne par le nombre très élevé d'électrons (6 trillions).

Maintenant que j'ai brièvement évoqué les grandeurs électriques telles que la tension électrique et le courant électrique, il me manque encore la résistance électrique. Afin que cela ne devienne pas trop aride et ne soit pas une récitation de connaissances théoriques, j'aborderai la résistance électrique dans les chapitres où elle joue un rôle important. C'est par exemple le cas dans le montage n° 1 sur la commande d'une diode lumineuse, car ce composant électronique qui fait office de voyant ne doit pas fonctionner sans limitation de courant. Et cette fonction de limitation du courant est assurée par un composant électrique précis. Nous y reviendrons plus tard.

Différents types de signaux

Je vous ai déjà montré précédemment les entrées et sorties de la carte Arduino Uno. Celles-ci sont mises à disposition par des *headers* (en-têtes en français) que vous pouvez voir sur la figure ci-après :

Figure 10 ▶
Les headers de la carte Arduino Uno

Vous pouvez connecter aux en-têtes de petits câbles patch ou directement des composants. Il existe différentes catégories de connexion sur lesquelles j'aimerais revenir en détail. Mais avant, je dois faire un petit détour pour expliquer la différence entre signaux analogiques et numériques.

Dans la nature, on trouve des formes de signaux aux tracés continus, comme la température ou la luminosité. Le tracé du signal d'un son aura une forme sinusoïdale sur un oscillographe, comme le montre cette figure :

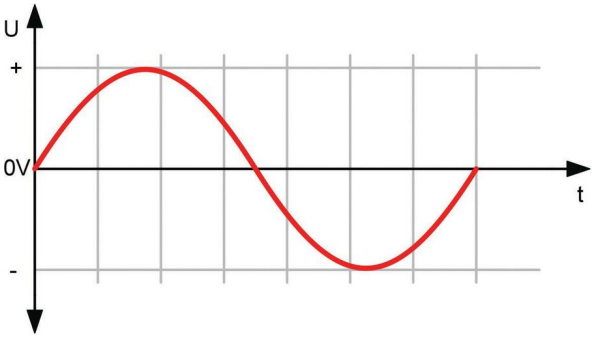


Figure 11
Une courbe sinusoïdale (analogique)

Il n'y a pas d'interruption au cours du temps, représenté sur l'axe horizontal *t*, et les valeurs se succèdent en permanence. Nous aurions du mal à représenter un signal de ce type sur un ordinateur qui ne connaît que les états *actif* ou *inactif*, qui correspondent aux niveaux de tension HIGH (5 V d'une liaison série TTL) et LOW (0 V). Le tracé pourrait ressembler à celui-ci :

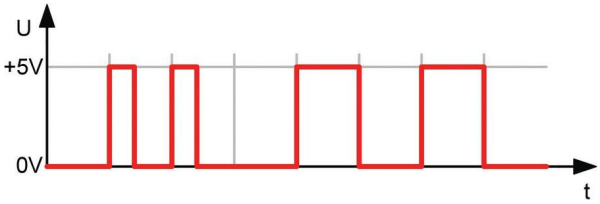


Figure 12
Une courbe numérique

Sur la carte Arduino comme sur d'autres microcontrôleurs, les variations de tension analogiques sont soumises à des restrictions sur lesquelles nous reviendrons plus loin.

Les entrées et sorties numériques

La carte Arduino Uno dispose de diverses entrées et sorties numériques. Les prises sont désignées par des dénominations que vous pouvez voir sur la figure ci-dessous :

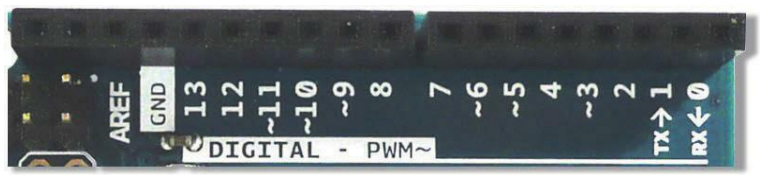


Figure 13
Les entrées et sorties numériques

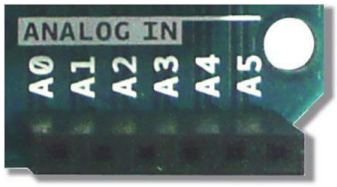
Les prises sont numérotées de droite à gauche de 0 à 13, certains chiffres étant précédés d'un *tilde*. Nous y reviendrons très vite. Il y a aussi deux

prises désignées par les lettres *RX* et *TX*. Elles ne doivent pas être utilisées pour la transmission de commandes, car elles sont affectées d'office à l'interface sérielle. La prise désignée par les lettres *GND* (Ground) correspond à la masse. Peut-être vous demandez-vous maintenant quels ports appartiennent à la catégorie des entrées et lesquels aux sorties. La réponse est simple. Les ports numériques peuvent être configurés librement par le biais de la programmation et servir aussi bien d'entrées que de sorties. Si une prise doit servir d'entrée, il faut toujours vérifier que la tension ne dépasse pas 5 V ! Sinon le microcontrôleur pourrait être définitivement endommagé.

Les entrées et sorties analogiques

Venons-en aux entrées et sorties analogiques. Les entrées analogiques sont illustrées sur la figure 14 :

Figure 14 ▶
Les entrées analogiques



Les dénominations vont de A0 à A5, soit un total de 6 entrées analogiques. Là aussi, les valeurs de tension doivent être comprises entre 0 V et 5 V. Ne perdez pas votre temps à rechercher des sorties analogiques, car vous n'en trouverez pas. Peut-être penserez-vous qu'elles ont été oubliées. Pas tout à fait. Elles se trouvent à un autre endroit et elles n'ont pas exactement le comportement que l'on attendrait de leur part.

Nous en arrivons maintenant aux ports numériques dont la dénomination est précédée d'un tilde. Il s'agit des ports D3, D5, D6, D9, D10 et D11 qui peuvent être configurés en tant que sorties analogiques. Des sorties numériques devraient donc faire office de sortie analogiques ?! Cela paraît un peu étrange.

C'est là que la *MLI* entre en scène. MLI est le pendant français de PWM (*Pulse-Width-Modulation*), autrement dit *modulation de largeur d'impulsion*. Nous avons ici affaire à un signal numérique et non à un signal analogique. Cela ne paraît pas très logique à première vue. Un signal MLI est un signal de fréquence et d'amplitude de tension constantes. La seule chose qui varie est le *rapport cyclique*.

Quand la fréquence f est constante, cela signifie que la durée d'une période T l'est aussi. Le seul aspect qui peut varier est la durée d'impulsion t.

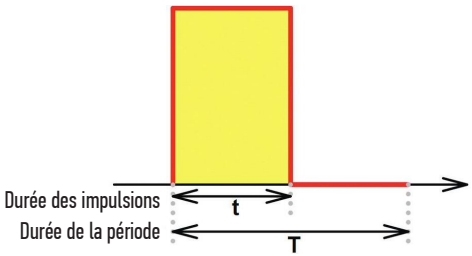


Figure 15
Durée d'une impulsion et durée d'une période au cours du temps

Plus l'impulsion est large, passant ainsi quasiment d'une droite unidimensionnelle à une surface, plus grande est l'énergie qui est transmise au consommateur. Examinons quatre possibilités.

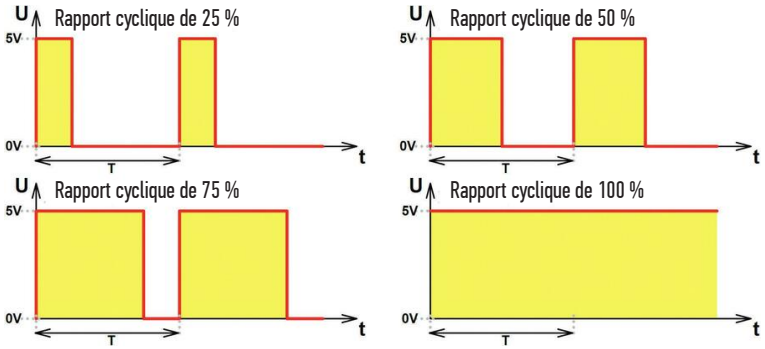


Figure 16
Exemples de MLI

Nous examinerons la commande en détail plus loin.

L'alimentation interne

Le connecteur d'alimentation comporte les broches ayant les fonctions suivantes :

Désignation	Fonction
VIN	Tension d'entrée de la carte Arduino pour une alimentation électrique externe 5 V
5 V	Sortie 5 V régulée
3V3	Sortie 3,3 V régulée. Le courant maximum est de 50 mA. GND
GND	Masse
IOREF	Tension de référence des ports d'E/S ou du microcontrôleur

Tableau 2
Broches d'alimentation en tension

Le bouton de réinitialisation

Le *bouton de réinitialisation* permet de redémarrer le microcontrôleur. Le sketch téléchargé n'est pas supprimé ; il est simplement redémarré.

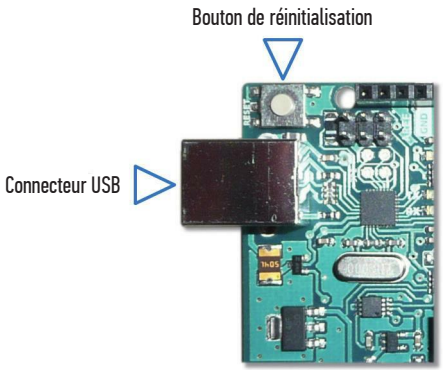


Figure 17 ▶ Le bouton de réinitialisation

L'interface sérielle

L'interface sérielle offre une autre possibilité de communication avec la carte Arduino. Elle est accessible via le port USB et peut être interrogée par l'intermédiaire d'un programme de terminal, tel que PiTTY. Nous n'avons pas besoin de logiciel supplémentaire, car le *moniteur série* inclus dans l'environnement de développement Arduino suffit. Il affiche des valeurs pendant la durée d'exécution du sketch, ce qui est non seulement très utile pour la surveillance de certaines valeurs de capteur, mais aussi pour une éventuelle recherche d'erreurs. Des fonctions plus avancées, comme l'échantillonnage ou l'affichage de valeurs mesurées qui sont transmises à d'autres programmes, peuvent être réalisées via l'interface sérielle. Le langage de programmation *Processing* est très utile dans ce contexte.

Informations relatives à l'interface sérielle et au port USB

Froncez-vous les sourcils à la seule mention de l'interface sérielle et de la communication par ce biais ? En effet, la carte Arduino est uniquement raccordée à l'ordinateur via un port USB et il faudrait maintenant qu'elle communique par l'interface sérielle ? Comment est-ce possible ? Cela s'explique par des raisons historiques : la première carte Arduino était connectée à l'ordinateur par l'interface sérielle RS232, car il n'y avait pas de port USB. Les modèles suivants ont été dotés d'une puce FTDI *(Future Technology Devices International)* qui permettait d'utiliser un port USB tel qu'une interface sérielle. Après l'installation du pilote correspondant, un port COM supplémentaire est alors disponible pour la carte Arduino. À la place de la puce FTDI (FT232RL), la carte Arduino Uno est dotée d'un microcontrôleur supplémentaire, l'*ATmega8U2*. Cette puce, en plus d'être

librement programmable, présente l'avantage de pouvoir être utilisée universellement comme périphérique USB, de la même façon qu'un clavier ou une souris. Vous trouverez des informations sur ce thème et d'autres aspects intéressants à l'adresse suivante :

https://learn.adafruit.com/arduino-tips-tricks-and-techniques/arduino-uno-faq

Les différentes mémoires

J'ai cité plus haut divers types de mémoire :

- flash ;
- SRAM ;
- EEPROM.

Il est important de connaître les différences et les domaines d'application de chacune. Vous trouverez de plus amples informations à l'adresse suivante :

https://www.arduino.cc/en/Tutorial/Memory

La mémoire flash

Un programme, ou *sketch* dans l'environnement Arduino, doit être stocké ou enregistré dans le microcontrôleur. Le sketch indique au microcontrôleur ce qu'il doit faire et les tâches qui doivent être exécutées. Un programme se divise en étapes (ou commandes) qui sont exécutées dans un ordre précis. La mémoire flash se charge du stockage. Même si la carte Arduino n'est plus raccordée à l'alimentation électrique, le sketch qui a été transféré sur le microcontrôleur reste en mémoire. Ces informations sont à nouveau accessibles après une nouvelle connexion électrique.

La SRAM

SRAM est l'abréviation de *Static Random Access Memory*. Lorsqu'un sketch est nécessaire pour traiter par exemple des valeurs de mesure, ces valeurs doivent être conservées sous une forme quelconque à l'intérieur du microcontrôleur. On utilise à cette fin des *variables* qui occupent de la place dans certaines parties de la mémoire pour l'échange et la manipulation des données. Toutefois, ces zones de mémoire sont volatiles, ce qui signifie que les informations disparaissent en cas de coupure de l'alimentation électrique. Cela n'a pas d'importance. En effet, le sketch n'utilise ces données que durant son exécution et elles sont recréées lors d'une nouvelle exécution. La zone de mémoire étant parfois très limitée sur certains modèles, il est alors possible de stocker des données dans la mémoire flash plutôt que dans la SRAM. Vous trouverez de plus amples informations à l'adresse suivante :

https://www.arduino.cc/en/Reference/PROGMEM

L'EEPROM

EEPROM signifie *Electrically Erasable Programmable Read-Only Memory*. Comme pour la mémoire flash, il s'agit d'une mémoire non volatile qui conserve les données qui y ont été enregistrées même en cas de coupure de l'alimentation électrique. Elle peut être utilisée pour enregistrer de façon permanente des données importantes, telles que des valeurs de mesure. Notez toutefois que les accès en écriture et en suppression à cette zone de mémoire sont limités à 100 000 cycles. Cette zone de mémoire n'est donc pas adaptée pour enregistrer des données échantillonnées lors de brèves mesures cycliques.

Vous venez de recevoir une grande quantité d'informations sur le matériel Arduino. La disposition des composants sur la carte Arduino Uno ne devrait plus avoir de secrets pour vous. Vous connaissez maintenant le microcontrôleur Atmega328, le cœur de la carte Arduino. Vous savez également comment la carte est alimentée en courant et quelles sont les entrées et sorties numériques et analogiques. Vous en savez un peu plus aussi sur les mémoires utilisées sur l'Arduino. Et surtout, vous savez ce que sont le courant et la tension et comment les représenter par des formules mathématiques.

Je vais vous montrer au chapitre 2 comment vous allez pouvoir piloter la carte Arduino à l'aide de son environnement de développement logiciel.

Chapitre 2

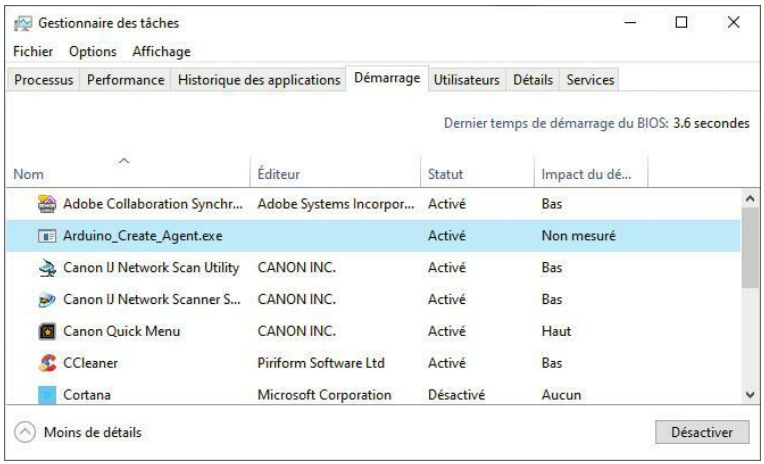
Arduino : le logiciel

Quand je parle d'Arduino, il s'agit aussi bien de la carte que du logiciel dont on se sert pour commander le microcontrôleur. Dans ce chapitre, vous apprendrez à installer les logiciels indispensables sur votre ordinateur. Nous passerons également en revue tout ce que vous devez savoir pour réussir à faire fonctionner correctement votre premier programme Arduino. Je vous montrerai la disposition de l'environnement de développement Arduino. Vous apprendrez aussi dans ce chapitre ce qu'il se passe à l'arrière-plan lorsqu'un programme se déroule sur votre carte Arduino. Je vous indiquerai enfin tous les composants nécessaires à votre atelier de développeur amateur.

Il existe théoriquement deux possibilités pour piloter votre carte Arduino. Vous pouvez recourir à un programme logiciel que vous installez sur votre ordinateur, l'environnement de développement Arduino en l'occurrence, ou vous pouvez utiliser Arduino Create, un éditeur basé sur le Web, à ouvrir avec votre navigateur.

Le plug-in à installer lors de l'utilisation d'Arduino Create basé sur le Web n'est pas un plug-in de navigateur mais un agent qui fonctionne en arrière-plan et qui configure son service quasiment indépendamment du navigateur. Tout navigateur utilise cet agent et doit pouvoir ensuite établir une connexion au cloud ou à Arduino. Cet agent redémarre à chaque redémarrage du système. Les informations sur le gestionnaire de tâches sont disponibles sous l'onglet *Démarrage*. Vous pouvez y voir l'entrée *Arduino_Create_Agent* (voir figure 1 page suivante).

Figure 1 ▶
La configuration des programmes de démarrage sous Windows



Pour éviter que cet agent démarre avec Windows, placez l'état sur *Désactivé*.

IDE Arduino ou Arduino Create ?

Si vous avez opté pour une installation locale de l'environnement de développement Arduino, tout votre code, y compris les bibliothèques, se trouve sur votre ordinateur. Vous avez donc tout sous la main et vous pouvez y accéder directement sans avoir besoin d'une connexion Internet. Lorsqu'il y a de nouvelles bibliothèques, autrement dit des programmes Arduino déjà créés, ou de nouvelles versions de bibliothèques existantes, il relève de votre responsabilité de mettre à jour votre système ou de les installer. Vous recevez évidemment des informations sur l'existence d'éventuelles mises à jour.

À l'inverse, la plate-forme Arduino Create vous propose d'enregistrer vos codes dans l'*Arduino IOT Cloud*, qui est une mémoire située quelque part sur Internet. Une connexion Internet continue est donc obligatoire, car tout est géré via un accès du navigateur web. L'avantage de cette version est que vous avez un accès partout dans le monde à toutes les ressources Arduino, codes et bibliothèques. Les bibliothèques proposées sont ainsi maintenues à un état actuel par un organe central et vous n'avez plus besoin de les mettre à jour. Je propose que vous testiez les deux versions. Vous pourrez ainsi opter pour celle qui vous convient le mieux. Aussi bien l'IDE Arduino qu'Arduino Create permettent de travailler de façon pratique. J'utilise dans ce livre l'IDE Arduino, parce que je préfère cette version et qu'elle ne m'oblige pas à avoir toujours une connexion Internet, bien que ce soit presque toujours le cas de nos jours. Personnellement, je n'aime pas les solutions sur le cloud.

Présentation de l'environnement

Au démarrage de l'environnement de développement Arduino que vous avez installé sur votre ordinateur, la fenêtre représentée sur la figure 2 s'ouvre : l'environnement de développement contient différentes zones. Vous y trouverez une zone d'entrée et une autre de sortie, des éléments de commande, ou encore des informations à propos des paramètres de connexion. Nous allons les examiner en détail.

L'éditeur

La zone dans laquelle vous saisissez votre code de programmation s'appelle *l'éditeur*. Il prend en charge la fonctionnalité de *syntax highlighting*, ce qui signifie que les mots-clés du langage de programmation sont généralement affichés en couleur.

Ligne d'informations

En dessous de l'éditeur se trouve une zone étroite dans laquelle s'affichent des informations relatives à la progression d'une opération ou des messages sur la dernière action exécutée. Vous pouvez voir sur la figure 2 que je viens de télécharger un sketch sur la carte Arduino.

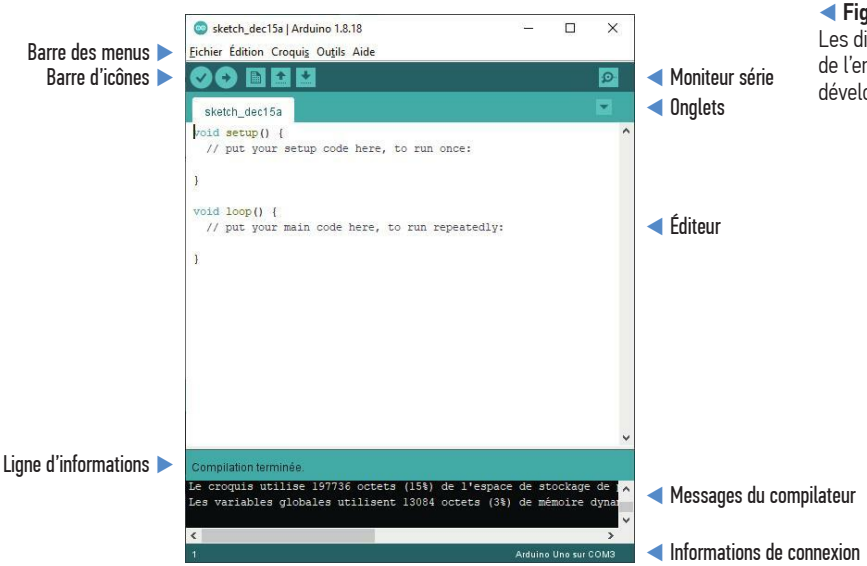
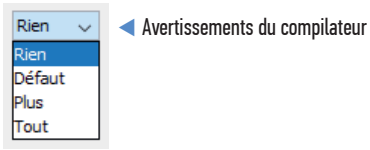


Figure 2
Les différentes zones de l'environnement de développement Arduino

Messages du compilateur

Lorsque le compilateur, c'est-à-dire le programme Arduino interne qui traduit votre code de programme dans un langage machine compris par le microcontrôleur, remplit son rôle de traduction du code source, la zone noire affiche des messages de progression ou signale d'éventuels problèmes. Enfin, l'utilisation de la mémoire y est aussi indiquée. Pour que le compilateur soit plus lisible, vous pouvez adapter le flux de langage avec l'option de menu *Fichier* | *Préférences* | *Avertissements du compilateur*.



L'option est réglée par défaut sur *Rien*.

Informations de connexion

Dans le bord inférieur de l'environnement de développement, vous pouvez voir les informations de connexion, c'est-à-dire la carte actuellement raccordée à un port COM. C'est utile en cas de problème de connexion ou pour la résolution d'erreurs.

Onglets

La barre d'onglets permet d'afficher les fichiers de codes sources d'un même sketch. Nous nous en servirons lorsque nous programmerons nos propres bibliothèques et catégories. Un seul sketch par environnement de développement peut être exécuté. Plusieurs sketches différents ne peuvent pas se trouver dans une instance de l'environnement de développement, contrairement aux onglets d'un navigateur qui est capable de gérer plusieurs connexions.

Moniteur série

L'icône de la loupe permet d'ouvrir le moniteur série qui se sert de l'interface sérielle pour afficher ou transmettre des informations.

Barre de menus

Comme dans d'autres programmes, la barre de menus permet d'accéder aux différentes commandes servant à gérer l'application.

Barres d'icônes

La barre d'icônes contient quelques boutons essentiels qui sont associés à des actions que vous employez fréquemment, comme la traduction (compilation) du code source, le téléchargement du code traduit sur le microcontrôleur ou l'ouverture et l'enregistrement des sketches.

Raccordement de la carte Arduino

Voyons maintenant comment raccorder la carte Arduino à l'ordinateur. La carte Arduino Uno possède le port USB suivant :

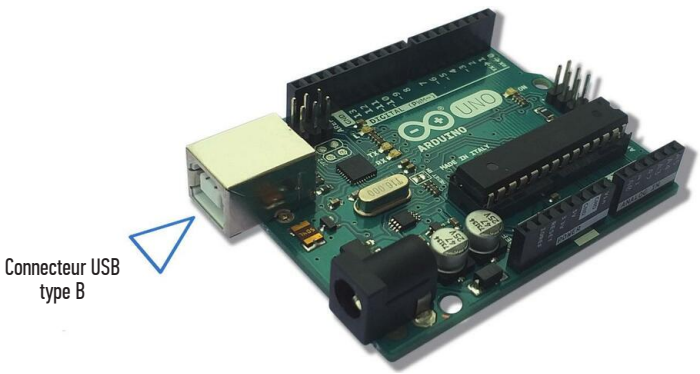


Figure 3
Le port USB
de la carte Arduino

L'ordinateur a besoin d'un port standard de *type A*. Vous trouverez partout un câble pour le raccordement, j'en ai une caisse pleine chez moi. Une fois la connexion établie, ce port USB remplit deux rôles :

- il sert à l'alimentation électrique de la carte Arduino ;
- il sert à la communication entre la carte et l'ordinateur.

Vous trouverez des informations détaillées sur les différents ports USB à l'adresse Internet suivante :

https://fr.wikipedia.org/wiki/USB

Lorsqu'une carte Arduino flambant neuve est alimentée en électricité, la LED sur la carte (désignée par un *L*) s'allume toutes les secondes sur la carte.

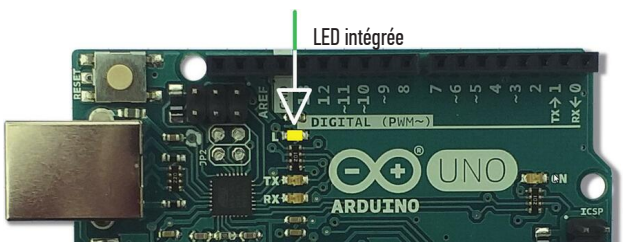


Figure 4
La LED L sur la carte

C'est ce qui se passe lorsqu'un sketch de base est préinstallé. Nous pouvons vérifier la communication avec la carte Arduino à l'aide de ce sketch car l'IDE dispose par défaut de nombreux sketches déjà installés.

Testons la communication entre l'ordinateur et Arduino

Une fois que la carte Arduino a été reconnue par l'IDE, nous allons tester la communication entre l'ordinateur et la carte Arduino. J'emploie le mot « communication » pour désigner le transfert d'un sketch (rappelez-vous : c'est ainsi que l'on appelle un programme dans l'environnement Arduino) sur le microcontrôleur. Nous allons donc ouvrir le sketch Blink (clignoter) dans l'IDE afin de modifier la fréquence du clignotement de la LED. Le programme sera ensuite compilé et transmis à la carte Arduino. Les différentes étapes énumérées ci-après présupposent que l'installation a réussi et que la carte Arduino a été reconnue au démarrage de l'IDE.

Étape 1 : ouvrir le sketch exemple

Sélectionnez l'option de menu *Fichier* | *Exemples* | *01.Basics* | *Blink* pour ouvrir le sketch dans l'IDE. Il a l'apparence suivante (j'ai supprimé toutes les lignes de commentaires contenant des informations complémentaires) :

```
void setup() {
 // initialize digital pin LED_BUILTIN as an output.
 pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
 digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the
// voltage level)
 delay(1000); // wait for a second
 digitalWrite(LED_BUILTIN, LOW); // turn the LED off by making the
// voltage LOW
 delay(1000); // wait for a second
}
```

Nous examinerons la signification des différentes lignes de code dans le montage n° 1. Leur rôle est de commander une diode LED qui se trouve sur la carte et qui doit clignoter avec un intervalle d'une seconde. Nous commencerons par vérifier que le sketch a bien été transféré par l'IDE à la carte Arduino. J'ai mentionné plus haut qu'une carte neuve est livrée

avec un sketch Blink préenregistré. Nous allons le modifier de façon à ce que la LED identifiée *L* sur la carte se mette à clignoter. Pour ce faire, nous allons modifier légèrement les deux lignes de code qui contiennent l'instruction `delay`.

Étape 2 : modifier le sketch exemple

La valeur qui est indiquée entre parenthèses définit la durée de la pause dans le traitement du programme. La valeur 1 000 désigne le délai en millisecondes (ms) sachant que 1 000 ms = 1 seconde. Par conséquent, lorsque la commande `delay` est atteinte, l'exécution marque une pause d'une seconde avant l'exécution de la commande située à la ligne suivante. Si nous changeons la valeur 1000 en 100, la vitesse de clignotement sera 10 fois plus rapide. Les deux lignes dans lesquelles figure la commande doivent être modifiées comme suit :

```
delay(100);
```

Le sketch modifié va – bientôt – pouvoir être transféré sur la carte Arduino. Je dis *bientôt*, car il faut encore faire une dernière vérification.

Étape 3 : choisir la bonne carte et le bon port COM

Comme Arduino (je parle maintenant de la société et non de la carte) a créé de nombreux modèles de cartes, nous devons préciser à l'environnement de développement le nom de la carte utilisée. Il faut également indiquer le port COM correct. Ces deux réglages s'effectuent à l'aide des commandes suivantes :

- *Outils | Type de carte ;*
- *Outils | Port.*

Dans mon cas, il s'agit *d'Arduino Uno* et de *COM5*.

Étape 4 : compiler et télécharger le sketch

Nous en arrivons à l'étape de la programmation qui se nomme la *compilation*. Procédons dans l'ordre. Pour représenter un programme informatique sous une forme plus ou moins lisible par l'homme, on a imaginé des langages dits évolués comme C/C++, Java et C#, pour ne citer qu'eux. Les commandes du sketch Blink sont écrites en anglais, comme la majorité des langages de programmation. L'intitulé de la commande décrit de façon plus ou moins claire la fonction qu'elle remplit. Ainsi, la commande `delay` pourrait être traduite par *retard* ou *avance* dont le sens ne nécessite pas de plus amples explications. Toutefois, le microcontrôleur ne sera pas plus avancé et nous devons lui donner un coup de pouce. Pourquoi ? Un microcontrôleur, comme n'importe quel processeur, ne reconnaît que les

commandes exprimées dans sa langue maternelle, le langage machine ou *Native Language*, qui varie en fonction des types de microcontrôleurs. Une séquence de traduction préalable est donc nécessaire pour traduire notre langage dit évolué (le C++ pour Arduino) en un langage machine compréhensible par le microcontrôleur. Cette tâche est accomplie par un *compilateur*.

QU'EST-CE QU'UN COMPILATEUR ?

Un compilateur est un programme qui traduit le code source d'un langage évolué comme C++ dans un langage que le processeur ou le microcontrôleur est en mesure de comprendre.

L'environnement de développement Arduino dispose, comme on l'a déjà évoqué, de plusieurs petites icônes avec des actions cachées. Le tableau ci-dessous vous indique les deux icônes les plus importantes et leurs fonctions. Les autres icônes sont bien sûr importantes, mais ne sont pas aussi essentielles que celles-ci :

Tableau 1 ▶ Quelques icônes importantes pour commencer

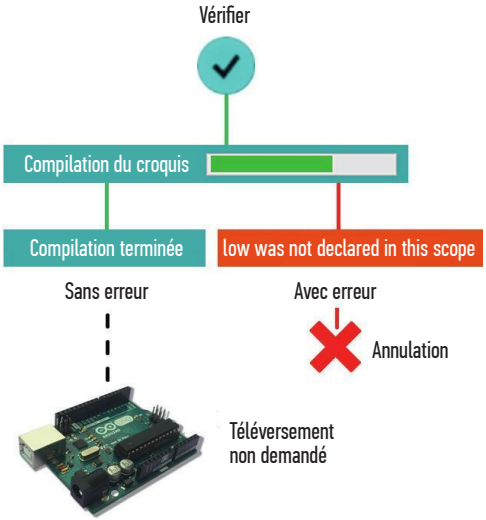
Icône	Fonction
	Vérification du code source par compilation
	Transmission au microcontrôleur lorsque la compilation est terminée

La première icône vérifie la syntaxe du *code source*, le texte affiché dans l'éditeur d'un environnement de développement. Qu'entend-on par « vérifier la syntaxe » ? Chaque langage possède différentes règles de syntaxe. Leur non-respect engendre des malentendus tant lors de la communication entre êtres humains qu'avec les processeurs. Si les commandes ne sont pas écrites ou formulées correctement, le compilateur signale immanquablement une erreur et interrompt le processus de traduction avec un message d'erreur correspondant. La figure 5 ci-contre montre ce qu'il se passe lorsque vous avez cliqué sur l'icône *Vérifier*.

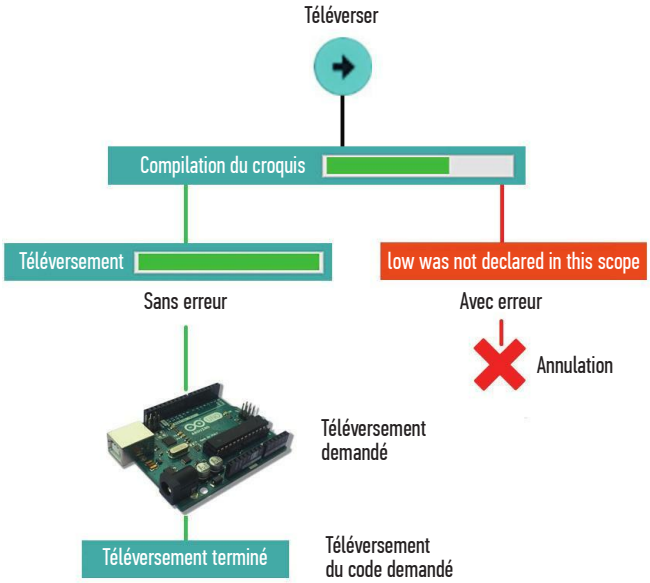
Au début, la compilation démarre en affichant la progression à l'aide d'une barre de progression verte. Si aucune faute de syntaxe n'a été trouvée, le processus se termine avec le message indiquant que la compilation est finie. Le chargement du code machine vers le microcontrôleur de la carte Arduino Uno ne s'est pas encore opéré. Cependant, si une erreur a été détectée dans le code du sketch, le processus de compilation s'interrompt avec un message d'erreur correspondant.

La deuxième icône analyse également le code source avec les mêmes procédures que pour la vérification, avec pour différence toutefois que le code machine est téléchargé sur le microcontrôleur de la carte Arduino

Uno après une compilation sans erreur. Le déroulement du processus est représenté sur la figure 6 :



◀ Figure 5
Déroulement après un clic sur l'icône Vérifier



◀ Figure 6
Déroulement après un clic sur l'icône Téléverser

Au début, la compilation démarre en affichant la progression à l'aide d'une barre de progression verte. Si aucune faute de syntaxe n'a été trouvée, le processus de téléchargement sur le microcontrôleur démarre et affiche également une barre de progression verte. Une fois le téléchargement

réussi, un message indique que le téléchargement est terminé. En présence d'une erreur dans le code du sketch, le processus de compilation s'interrompt avec un message d'erreur correspondant, comme dans le cas de la vérification.

Il est également possible de choisir immédiatement le téléchargement sans passer par la compilation préalable, car une compilation préalable y est également exécutée. Pendant le processus d'*upload* (téléchargement), certaines LED sur la carte Arduino s'allument. Regardons cela de plus près :

Figure 7 ▶
Trois LED importantes sur la carte Arduino

Trois LED se trouvent à gauche du nom de la carte. Celle qui est désignée par un *L* indique l'état de la broche numérique 13. Un peu plus bas, vous pouvez voir deux autres LED désignées par les lettres *TX* (Transmettre) et *RX* (Recevoir). Il s'agit de l'affichage d'état de l'interface sérielle qui relie la carte Arduino à l'ordinateur. Lors du téléchargement, des informations en langage machine sont transmises via cette interface. La carte signale l'exécution de ce processus par le clignotement irrégulier de ces deux LED. Quand le transfert est terminé, quelle qu'en soit la raison, les diodes s'éteignent. Cela permet de contrôler visuellement le processus de téléchargement. Une fois le téléchargement réussi, la diode *L* clignote plus rapidement, ce qui indique que la modification du sketch a bien été prise en compte. Sur le côté droit de la figure 7, vous pouvez voir une autre LED dénommée *ON*. Elle s'allume lorsque la carte est sous tension.

LE CODE MACHINE DE LA CARTE ARDUINO

Cet encadré n'a pas de rapport direct avec la programmation de la carte Arduino, mais il est important que vous compreniez les étapes qui se déroulent en arrière-plan. Je vous ai déjà un petit peu parlé de l'environnement de développement, du compilateur et des langages de programmation C/C++. Comment se déroule la compilation et qu'est-ce qui est au juste transféré sur le microcontrôleur de la carte Arduino ?

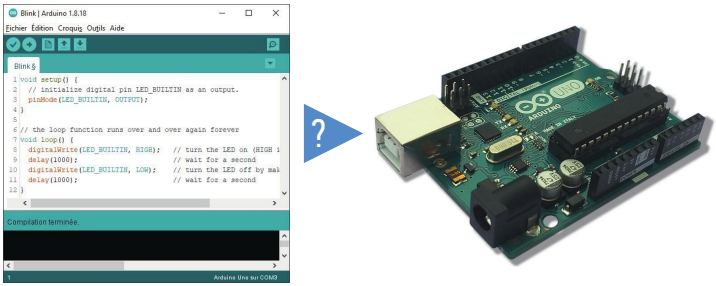


Figure 8
Que se passe-t-il en arrière-plan lors du transfert du sketch sur la carte Arduino ?

Nous pouvons diviser ce déroulement en quelques étapes logiques :

Étape 1

Une vérification du code du sketch est faite par l'environnement de développement, afin de garantir que la syntaxe C/C++ est correcte.

Étape 2

Le code est ensuite envoyé au compilateur (dont le nom est AVR-GCC), qui le transcrit en un langage lisible par le microcontrôleur : c'est le ***langage machine***.

Étape 3

Le code compilé fusionne avec certaines bibliothèques Arduino qui apportent les fonctionnalités de base, ce qui aboutit à la création d'un fichier au format *Intel HEX*. Il s'agit d'un fichier texte qui contient des informations binaires pour le microcontrôleur. Ci-dessous un court extrait du premier sketch :

Blink.hex

```
1  :10000000 0C945C00 0C946E00 0C946E00 0C946E00 CA
2  :10001000 0C946E00 0C946E00 0C946E00 0C946E00 A8
3  :10002000 0C946E00 0C946E00 0C946E00 0C946E00 98
4  :10003000 0C946E00 0C946E00 0C946E00 0C946E00 88
5  :10004000 0C941301 0C946E00 0C946E00 0C946E00 D2
6  :10005000 0C946E00 0C946E00 0C946E00 0C946E00 68
7  :10006000 0C946E00 0C946E00 00000000 00240027 0029
8  :10007000 2A000000 00000025 00280028 002B0004 04040404 CE
9  :10008000 04040404 02020202 02020303 03030303 0342
10 :10009000 01020408 10204080 01020408 10200102 1F
11 :1000A000 04081020 00000008 00020100 00030407 FB
```

Figure 9
Extrait d'un fichier Intel-HEX

Le microcontrôleur comprend ce format, car c'est son *Native Language*, c'est-à-dire sa *langue maternelle*.

Étape 4

Le chargeur d'amorçage (*bootloader*) transmet le fichier Intel HEX via la liaison USB à la mémoire flash du microcontrôleur. Ledit *processus de téléversement*, donc la transmission sur la carte, est assuré par le programme **avrdude**, qui fait partie intégrante de l'installation Arduino. Vous le trouverez dans le répertoire `Program Files (x86)\Arduino\hardware\tools\avr`.

La gestion des bibliothèques

L'environnement de développement dispose d'un grand nombre de bibliothèques qui peuvent être installées sur votre ordinateur. Elles peuvent être utilisées à différentes fins. Des *mises à jour* sont régulièrement proposées. Un message s'affiche pour vous en informer :

Figure 10 ▶
Message avertissant de la présence de mise à jour des bibliothèques

Lorsque vous cliquez sur le lien indiqué *Bibliothèques*, une nouvelle fenêtre s'affiche, dans laquelle les nouvelles mises à jour peuvent être sélectionnées. La liste des bibliothèques installées est mise à jour, ce qui est représenté par la barre de progression ci-dessous :

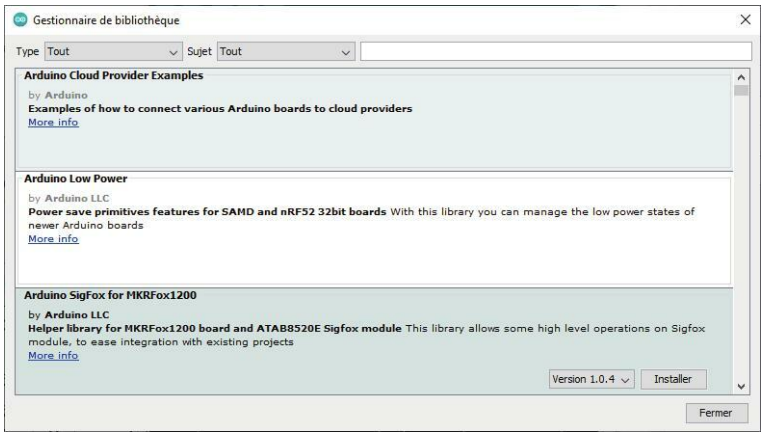


Figure 11 ▶
La liste des mises à jour disponible

Pour installer une nouvelle bibliothèque, passez la souris sur la bibliothèque correspondante. Deux messages s'affichent en règle générale. Une liste vous propose les versions disponibles pour une installation ultérieure, le bouton *Installer* s'affiche :

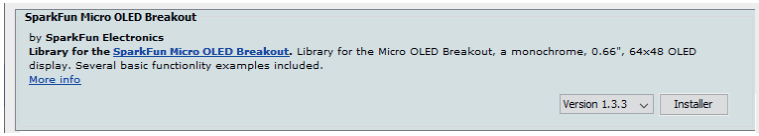


Figure 12 ▶
Installation d'une nouvelle bibliothèque

Une bibliothèque déjà installée est identifiée par l'inscription verte *INSTALLED*, et si une mise à jour est disponible, le bouton *Mise à jour* apparaît au passage de la souris :

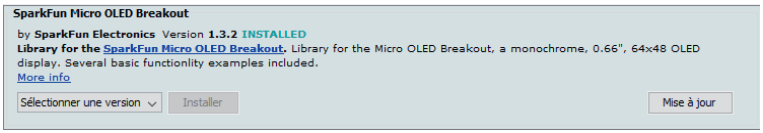


Figure 14
Le bouton Mise à jour s'affiche.

La gestion des bibliothèques est également accessible par l'option de menu *Outils* | *Gérer les bibliothèques* qui permet d'ouvrir la fenêtre ci-dessous.

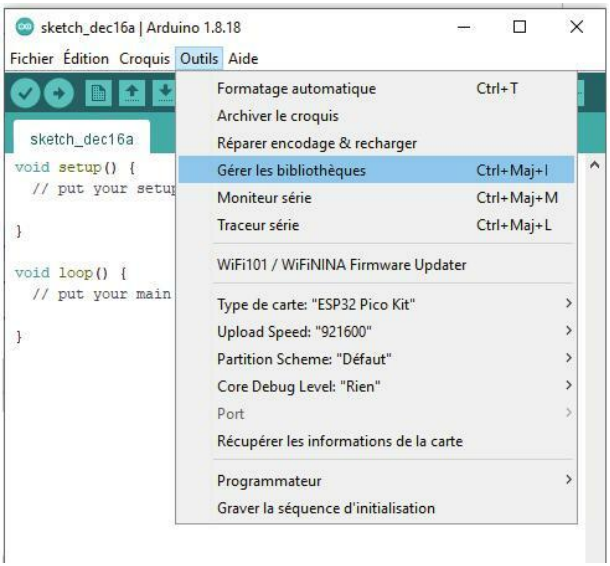
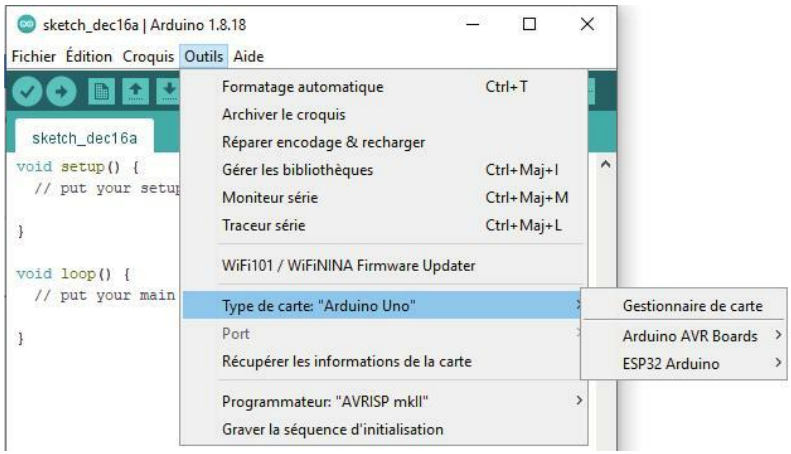


Figure 14
Ouvrir la fenêtre de gestion des bibliothèques

La gestion des cartes

L'environnement de développement Arduino peut non seulement programmer et contrôler des cartes Arduino, mais aussi celles d'autres fabricants, par exemple les cartes ESP32 et ESP8266 très connues et appréciées. Le gestionnaire de cartes disponible sous l'option de menu *Outils* | *Cartes* | *Gestionnaire de cartes* sert justement à intégrer ces cartes dans l'environnement de développement.

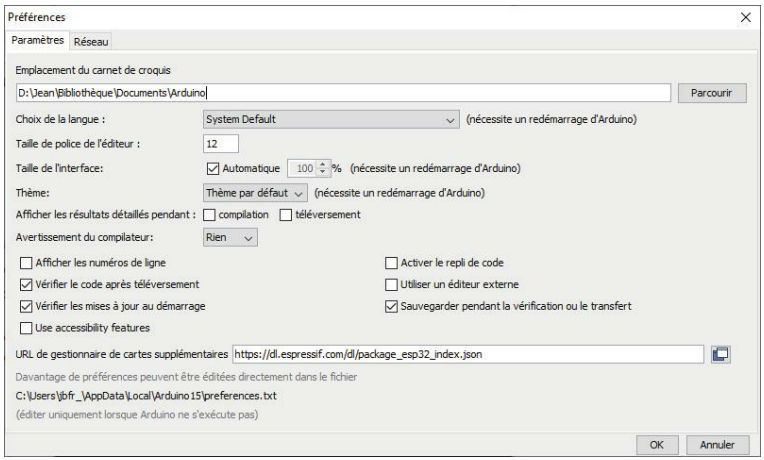
Figure 15 ▶
Ouvrir la fenêtre de gestion des cartes



Si je veux par exemple ajouter la compatibilité à la carte *megaAVR* à mon environnement de développement Arduino, je tape en haut à droite le terme *megaAVR* et le pack d'installation correspondant s'affiche. Il peut ensuite être installé.

Figure 16 ▶
Ajouter une nouvelle carte

Cette recherche peut aussi s'appliquer à la gestion des bibliothèques. Vous êtes peut-être intéressé par les cartes ESP32 et ESP8266 et vous avez cherché le terme ESP32. Malheureusement, le terme recherché ne donne pas de résultats et la liste reste vide. Pas de panique ! L'explication est que de nombreuses cartes de fournisseurs tiers fournissent des informations et des fichiers d'installation dans un répertoire maison. Comment communiquer ces chemins d'accès à l'environnement de développement Arduino ? C'est très simple ! Dans l'option de menu *Fichier* | *Préférences*, ouvrez la fenêtre de dialogue correspondante. Celle-ci se présente comme sur la figure 17 page suivante. J'y ai déjà entré le chemin d'accès requis pour la compatibilité ESP32.



◀ Figure 17
Les Préférences d'Arduino

Le chemin d'accès à utiliser pour la compatibilité à la carte ESP32 est :

https://dl.espressif.com/dl/package_esp32_index.json

Pour ajouter la compatibilité de la carte ESP8266, ajoutez à l'URL précédente le chemin d'accès suivant (séparé par une virgule) :

http://arduino.esp8266.com/stable/package_esp8266com_index.json

Les deux cartes sont ensuite prêtes à être installées dans l'environnement de développement Arduino. Il existe une source d'extension non officielle pour des cartes de fournisseurs tiers, que vous trouverez à l'adresse ci-dessous :

https://github.com/arduino/Arduino/wiki/Unofficial-list-of-3rd-party-boards-support-urls

De très nombreuses cartes pouvant être programmées avec l'IDE Arduino y figurent. Avant de commencer, copiez les chemins d'accès affichés sur cette page dans les préférences Arduino dans la ligne que je vous ai indiquée.

Le code du sketch dans l'environnement de développement

Certes, je n'ai ni la place ni la prétention de vous proposer ici une formation de base en programmation C/C++. Toutefois, j'aimerais apporter quelques précisions sur l'emploi de ce langage dans l'environnement de développement. Le langage de programmation C/C++ fait la différence entre les majuscules et les minuscules : en anglais, on dit qu'il est *case sensitive*. Par conséquent, vérifiez la façon dont les commandes sont écrites. Prenons l'exemple de la commande `digitalWrite`. Elle se compose de deux mots (*digital* et *Write*). En programmation, la création d'un mot par

l'assemblage de termes dont l'initiale est en majuscule (sauf pour le premier mot) se nomme le *camel case*. Le code suivant sera donc considéré comme incorrect :

```
digitalwrite
```

Les erreurs de ce type sont détectées par l'environnement de développement Arduino et elles sont signalées. Un mot qui fait partie d'une commande est généralement affiché en couleur. S'il a été mal écrit, il n'apparaît pas en couleur et ne se distingue pas du reste du code qui reste noir. Dans l'environnement de développement, on voit que la commande de la ligne 8 est correcte et qu'elle présente un caractère de couleur correspondant. Comme la commande a été mal écrite à la ligne 9, elle n'a pas été détectée et elle est donc affichée en noir.

```
7  void loop() {
8    digitalWrite(13, HIGH); // Syntaxe correcte
9    digitalwrite(13, HIGH); // Syntaxe incorrecte
10 }
```

Problèmes courants

Il est inévitable que tôt ou tard, vous vous heurtiez à des problèmes. Personne n'est à l'abri, moi-même y compris. Mais je sais par expérience qu'il existe des sources d'erreurs récurrentes.

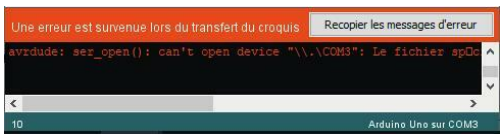
La carte est-elle sous tension ?

La carte ne sera pas alimentée si le câble USB est défectueux. Dans ce cas, la LED *ON* ne s'allumera pas. La carte ne sera pas non plus reconnue par le système et elle ne sera pas proposée sous la forme d'un port COM.

Ai-je bien choisi la carte et le port COM ?

Si la carte et le port COM n'ont pas été correctement choisis, le téléchargement dure très longtemps avant qu'un message d'erreur n'apparaisse en rouge dans la partie inférieure de l'environnement de développement :

Figure 18 ▶
Message d'erreur Arduino



Il n'est pas possible d'accéder au port COM3 configuré précédemment. Arduino propose une page de résolution des erreurs à l'adresse suivante :

https://arduino.cc/en/pmwiki.php?n=Guide/Troubleshooting#toc1

CHOISIR LE BON PORT COM

Si vous rencontrez des problèmes lors de la sélection du bon port COM, vous disposez d'une solution très simple. Regardez la liste des ports COM proposés dans l'option de menu *Outils* | *Port*. Débranchez ensuite le câble USB de la carte Arduino et regardez quel port n'est plus indiqué dans la liste. Rebranchez le câble pour voir quel port a été ajouté. L'environnement de développement Arduino indique tous les ports disponibles presque immédiatement après la modification.

CE QUE VOUS DEVRIEZ AVOIR SOUS LA MAIN DANS VOTRE ATELIER DE BRICOLAGE ARDUINO

Passons maintenant au matériel que nous allons utiliser pour tous les montages décrits dans cet ouvrage. Pour que les composants électroniques et électriques puissent être connectés correctement à la carte Arduino, deux types d'éléments sont indispensables : d'une part, des plaques de prototypage, ou *breadboards*, sur lesquelles les composants sont enfichés et, d'autre part, un nombre suffisant de câbles qui sont indispensables pour la connexion électrique entre les différents composants. Nous utilisons à cette fin des cavaliers flexibles qui se branchent facilement aussi bien sur la plaque de prototypage que sur la carte Arduino.

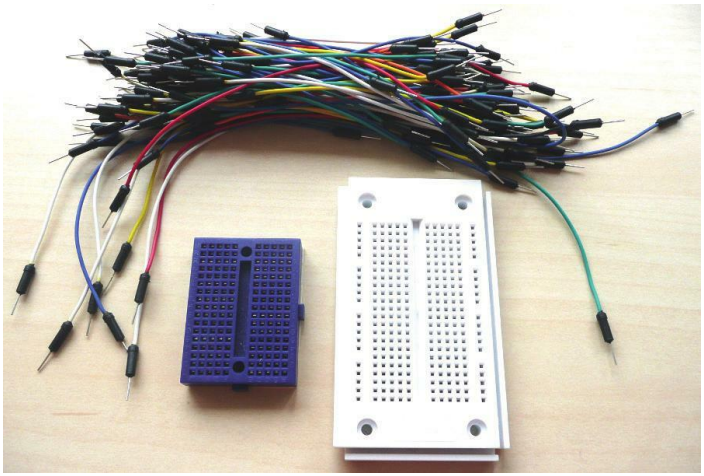


Figure 19
Différentes plaques de prototypage et de cavaliers flexibles

La figure 19 de la page précédente présente deux plaques de prototypage répandues qui existent en différentes tailles, formes et couleurs. Je vous recommande d'acheter une ou plusieurs plaques de différentes dimensions, car elles sont vraiment abordables. Investissez un peu plus en cas de doute dans une plaque supplémentaire, cela en vaut vraiment la peine. Les cavaliers flexibles, appelés également câbles patch, sont également bon marché. Il existe également de très jolies variantes combinées à une carte Arduino et une plaque de prototypage sur un support fixe sous forme de combinaison, comme l'illustre la figure 20. De cette façon, la carte Arduino et la plaque de prototypage sont indissociables, ce qui facilite dans une certaine mesure la mise en circuit. Par ailleurs, la figure 20 montre une carte Arduino Uno proposée comme clone et donc un peu moins chère que la carte originale. Les clones sont des copies proposées par d'autres sociétés. Et comme la carte Arduino Uno a été publiée avec une licence de matériel Open Source, chacun est libre de copier la carte et de la modifier. La carte n'a pas besoin d'être une carte originale car l'Open Source autorise aussi ce genre de cartes.

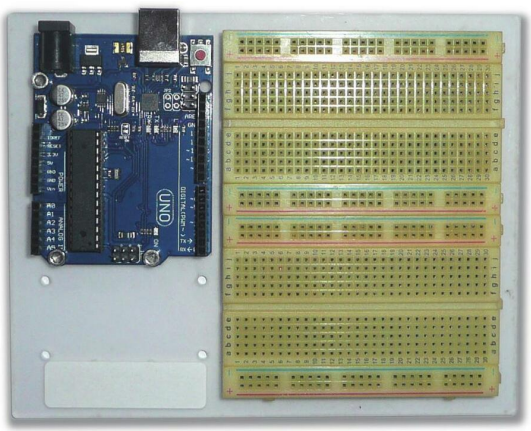


Figure 20 ▶
Combinaison d'une carte Arduino et d'une plaque de prototypage

Une solution très élégante est l'utilisation d'une carte faite maison, que j'ai baptisée *Arduino Discoveryboard*. Je vous présente cette carte en détail au chapitre 4 et vous montre comment la construire vous-même. L'Arduino Discoveryboard vous fera gagner beaucoup de temps, que vous pourrez alors consacrer à la construction des montages. Mais comme je l'ai déjà dit : vous n'êtes pas obligé de le faire tout de suite.

Un multimètre est également pratique pour mesurer des tensions, des courants et des résistances, ou simplement pour vérifier des connexions électriques.

Il existe d'autres instruments très intéressants que je ne peux pas tous vous présenter ici. Je vous ai présenté les *incontournables* qui ne sont pas très nombreux et que tout bricoleur devrait avoir sous la main. Le multimètre fait partie de ces instruments qui font gagner un temps précieux pour résoudre les erreurs. Que ce soit pour mesurer des tensions ou des courants comme contrôleur de continuité ou pour déterminer une résistance, vous devriez toujours avoir un multimètre sous la main.

Figure 21
Un multimètre

De l'idée jusqu'au téléchargement dans le microcontrôleur

Dans ce chapitre, nous avons appris à installer le logiciel nécessaire au développement d'un programme, puis à transférer un sketch sur le microcontrôleur après sa saisie dans l'environnement de développement Arduino. Cette procédure ne se limite pas à ce que nous avons pu entrevoir avec ce premier coup d'œil. Reprenons les différentes étapes en détail.

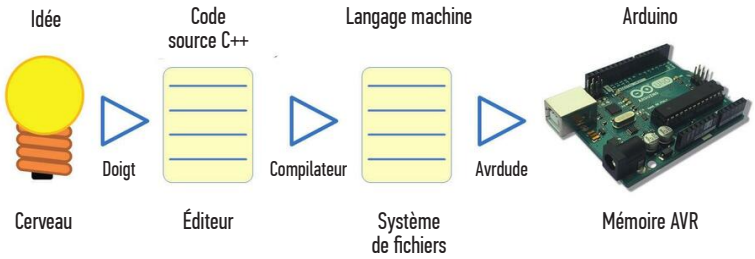


Figure 22
De l'idée à sa mise en œuvre

Au début, il y a toujours une idée qui naît et mûrit dans le cerveau. Comme le microcontrôleur ne peut évidemment pas lire les pensées, nous avons besoin d'outils. Nous nous servons d'un éditeur de texte qui fait partie de l'environnement de développement pour transposer nos pensées en actions qui sont formulées sous forme de commandes. Cette formulation emploie un vocabulaire qui est composé d'éléments du langage de programmation C++. Ces éléments sont affichés à l'aide d'un compilateur (AVR-GCC) qui est chargé de leur traduction dans un langage pouvant être

compris par le microcontrôleur. La transcription de ce langage machine est réalisée par un programme nommé `Avrdude` qui se charge de la transmission du code au microcontrôleur Atmega328, monté sur la carte Arduino.

Au cours de ce chapitre, vous avez découvert les outils logiciels que vous devez maîtriser afin de programmer la carte Arduino, soit par le biais du navigateur basé sur le Web soit par l'application distincte sur votre ordinateur. Je vous ai montré comment raccorder la carte à votre ordinateur et comment un sketch préinstallé fait immédiatement clignoter la LED intégrée. En changeant le code, vous avez pu modifier la cadence de clignotement. Je vous ai également indiqué quelques sources d'erreurs récurrentes dans le cas où vous rencontriez des problèmes de connexion entre l'ordinateur et la carte Arduino.

Le chapitre 3 aborde en profondeur les bases de la programmation.

Chapitre 3

N'ayons pas peur de la programmation : les bases du codage

Dans le chapitre 2 sur le logiciel Arduino, nous avons déjà un peu abordé la programmation. Je vous ai montré un premier programme, appelé sketch, sans préciser ce que programmer signifie. Je vais maintenant expliciter les notions de base de la programmation : l'algorithme, les variables, les constantes et les différents types de données. Et comme si cela ne suffisait pas, je vais vous donner en plus des explications sur les structures de contrôle et les fonctions. À la fin de ce chapitre, vous devriez connaître à peu près la palette de fonctions dont vous disposez pour écrire un programme.

Vous connaîtrez uniquement les bases du langage de programmation C++, dans lequel ont été créés les sketches Arduino. Si vous avez envie d'approfondir ce langage ou si vous souhaitez trouver des informations complémentaires, vous pouvez lire l'ouvrage *Programmer en C++ moderne* de Claude Delannoy (Éditions Eyrolles).

Pour programmer, vous avez évidemment besoin d'une machine, que ce soit un ordinateur ou un microcontrôleur comme la carte Arduino. Il est clair que ce genre d'appareil ne dispose pas d'une intelligence propre. Sans instructions données par un programme, il n'est rien d'autre qu'un matériel inutile, qui consomme tout au plus du courant. Matériel et logiciel sont dépendants l'un de l'autre et ne peuvent fonctionner qu'ensemble. Les programmes disent au matériel ce qu'il doit faire. Et le programmeur dit au logiciel ce que celui-ci doit dire au matériel.

Qu'est-ce qu'un programme, ou sketch ?

Pour ce qui est de la programmation, nous avons affaire en règle générale à deux éléments.

Élément de programmation 1 : l'algorithme

Pour que le sketch puisse exécuter de manière autonome une tâche définie, un algorithme est créé, lequel contient un certain nombre d'étapes indispensables à la réussite du sketch.

QU'EST-CE QU'UN ALGORITHME ?

Un algorithme est la description d'une action en vue de régler un problème et se compose d'un nombre important d'étapes devant se dérouler dans un certain ordre.

Un algorithme est une règle de calcul qui fonctionne à la manière d'une liste de contrôle. Imaginez que vous vouliez construire un boîtier en bois pour y loger votre carte Arduino, pour que cela soit un peu plus beau, plus ordonné et plaise également à vos amis. N'achetez pas de bois sans avoir préparé un plan répondant par exemple aux questions suivantes :

- Quelles sont les dimensions de la caisse ?
- De quelle couleur doit-elle être ?
- Où des ouvertures doivent-elles être pratiquées pour installer par exemple des interrupteurs ou des lampes ?

Après avoir réuni le matériel, passez à la fabrication proprement dite, en procédant étape par étape dans un certain ordre :

- fixation des plaques de bois ;
- mise à dimension des plaques de bois ;
- ponçage des bords avec du papier de verre ;
- perçage de certaines plaques de bois, appelées à recevoir des ports ;
- vissage des plaques de bois entre elles ;
- mise en peinture du boîtier ;
- insertion de la carte Arduino et câblage des interrupteurs ou des lampes.

Telles sont les étapes par lesquelles il faut passer pour arriver à vos fins. Il en va de même pour l'algorithme.

Élément de programmation 2 : les données

Vous avez très certainement soigneusement noté les dimensions sur le plan, de manière à pouvoir les consulter pendant la construction. Il faut faire en sorte que tout coïncide à la fin. Ces dimensions sont comparables aux données d'un sketch. L'algorithme utilise des valeurs temporaires qui l'aident dans son travail de prise en charge des différentes étapes. Il utilise pour cela une technique qui lui permet de stocker des valeurs et de les rappeler par la suite. Les données sont en effet sauvegardées dans des variables et disponibles à tout moment dans la mémoire. Vous en saurez bientôt plus. Nous y reviendrons plus tard.

QUE SONT LES DONNÉES ?

Les informations saisies et les valeurs mesurées sont généralement désignées comme étant des données. Elles peuvent se présenter sous forme de texte ou de valeurs numériques.

Qu'est-ce que le traitement des données ?

On entend par *traitement des données* l'utilisation d'un *algorithme* qui se sert de *données* en entrée pour en obtenir d'autres, modifie celles-ci par différents calculs et produit des résultats en sortie. Ce principe est appelé *IPO*.

Figure 1
Le principe IPO

Que sont les variables ?

Nous avons déjà vu que des données étaient sauvegardées dans des variables. Ces dernières jouent un rôle central dans la programmation et sont utilisées dans le traitement des données pour stocker des informations de toutes sortes.

QU'EST-CE QU'UNE VARIABLE ?

En programmation, une variable est un élément associant un nom à une valeur modifiable, à laquelle il est possible d'accéder pendant le déroulement d'un calcul et qui est maintenue en réserve dans la mémoire.

Une variable occupe dans la mémoire une certaine place qu'elle garde libre. L'ordinateur ou le microcontrôleur gère cependant cette mémoire (de travail) selon ses propres méthodes. Tout ceci se fait au moyen de désignations codées que tout un chacun a certainement du mal à retenir. C'est pour cette raison que vous pouvez doter les variables de noms évocateurs qui renvoient en interne aux adresses de mémoire concernées.

Sur la figure 2, la variable nommée `ledPin` pointe sur une adresse de départ dans la mémoire de travail. Elle peut également être considérée comme une sorte de référence renvoyant à quelque chose de particulier. La ligne suivante d'un sketch montre l'utilisation d'une variable nommée `ledPin` :

```
int ledPin = 13; // Variable déclarée + initialisée avec broche 13
```

à laquelle la valeur numérique 13 a été attribuée. Plus loin dans le sketch, cette variable est évaluée et continue d'être utilisée. Le terme `int` est l'abréviation du mot *Integer*. Il s'agit d'un type de donnée utilisé dans le traitement des données pour caractériser des nombres entiers, ce qui nous amène au point suivant.

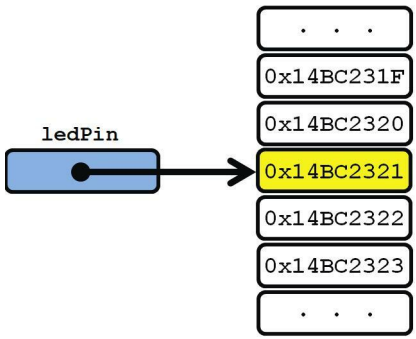


Figure 2 ▶
Variable ledPin pointant sur une zone de la mémoire de travail

Que sont les constantes ?

J'ai lu quelque part sur Internet qu'une variable constante est une variable dont la valeur ne peut plus être modifiée après initialisation. Relisez cette phrase et réfléchissez à ce qui suit :

- constant : non modifiable, qui persiste dans l'état où il se trouve ;
- variable : modifiable, non limité à une possibilité.

La première phrase semble être paradoxale. Qu'est-ce que cela peut vouloir dire ? La deuxième partie de la phrase est juste : une constante après son initialisation ne peut plus être modifiée pendant le déroulement du programme. Une constante est identifiée comme étant constante par le mot-clé `const`.

```
const int a = 17;
```

Les lignes de code ci-dessous seraient ainsi inadmissibles et entraîneraient un message d'erreur, étant donné que le contenu d'une constante ne doit plus être modifié :

```
const int a = 17; // Constante
void setup() {
a = 18; // Non autorisé !
}
...
```

Les types de données

Voyons maintenant les différents types de données et essayons de savoir ce qu'est exactement un type de données et pourquoi il en existe plusieurs types. Le microcontrôleur gère ses sketches et ses données dans sa mémoire. Cette mémoire est une zone structurée qui est gérée par des adresses et qui enregistre ou restitue des informations, lesquelles sont stockées sous la forme de 1 et de 0.

QU'EST-CE QU'UN TYPE DE DONNÉES ?

Un type de données décrit une certaine quantité d'objets de données (variables), qui ont tous la même structure. Chaque variable doit avoir un certain type de données.

La plus petite unité de mémoire logique est le *bit*, qui peut justement enregistrer les deux états 1 ou 0. Imaginez celle-ci comme une sorte d'interrupteur électronique, qui peut être allumé ou éteint. Comme un bit ne peut représenter que deux états, il est logique et nécessaire d'utiliser plusieurs bits pour enregistrer des données. Un ensemble de 8 bits forme

1 *octet* et permet d'enregistrer 2^8 = 256 états différents. S'agissant d'un système binaire, la base 2 est utilisée. Avec 8 bits, on peut donc couvrir un domaine de valeurs compris entre 0 et 255.

Voici pour commencer une liste des principaux types de données, auxquels vous serez confrontés à l'avenir :

Tableau 1 ▶ Type de données avec plages de valeurs correspondantes

Type de données	Plage de valeurs	Largeur de données	Exemple
void	aucune	zéro	`void setup() {}`
octet	0 à 255	1 octet	`byte valeur = 42;`
unsigned int	0 à 65.535	2 octets	`unsigned int secondes = 46547;`
int	–32.768 à 32.767	2 octets	`int ticks = -325;`
long	-2^{31} à $2^{31}-1$	4 octets	`long valeur = -3457819;`
float	$-3.4 * 10^{38}$ à $3.4 * 10^{38}$	4 octets	`float valeur de mesure = 27.5679;`
double	voir *float*	4 octets	`double valeur de mesure = 27.5679;`
boolean	*true* ou *false*	1 octet	`boolean flag = true;`
char	-128 à 127	1 octet	`char mw = 'm';`
String	variable	variable	`String nom = "Erik Bartmann";`
Array	variable	variable	`int pinArray[] = {2, 3, 4, 5};`

La plupart des types de données indiqués ici sont également utilisés dans ce livre.

Que sont les fonctions ?

Une fonction est, dans la plupart des langages de programmation, la désignation d'une construction permettant de structurer le code source de sorte que ces parties du programme, qui peuvent être qualifiées de sous-programmes, puissent être réutilisées dans le programme principal et ouvertes plusieurs fois à différents endroits. Pour la programmation orientée sur les objets, sur laquelle je reviendrai, il existe des constructions comparables qui portent le nom de *méthodes*. À cela s'ajoute également le terme d'*encapsulage*, qui se rapporte également aux fonctions. Un encapsulage est le récapitulatif ou la dissimulation de données ou d'informations, l'accès s'effectuant via une interface définie. Cette interface est représentée par l'ouverture de la fonction. Par exemple, si vous souhaitez représenter plusieurs fois dans un sketch la valeur moyenne de deux chiffres, les lignes de code suivantes sont normalement presque toujours nécessaires, bien que j'aie formulé tout cela de manière un peu complexe pour rendre le sens de la fonction plus claire. Il existe des fonctions qui fournissent à l'appelant

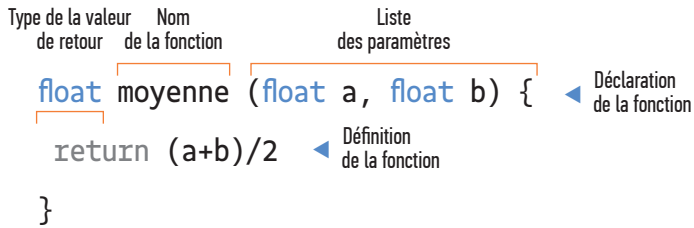
une valeur de retour, comme c'est le cas ici. Il y a aussi des fonctions qui exécutent une tâche sans renvoyer à l'appelant une valeur de retour.

```
float a = 5.4, b = 7.36;
float somme = a + b;
float moyenne = somme / 2;
```

Pour calculer la moyenne de deux chiffres, ceux-ci sont additionnés et le résultat est divisé par deux. Maintenant, si vous souhaitez créer cette moyenne à plusieurs endroits dans le sketch, vous devrez insérer à ces endroits les deux lignes ci-dessous. Mais cela devient plus facile avec la définition d'une fonction. Celle-ci pourrait être :

```
float moyenne(float a, float b)
{ return (a + b)/2;
}
```

Regardons cela de plus près car beaucoup de choses entrent en jeu :



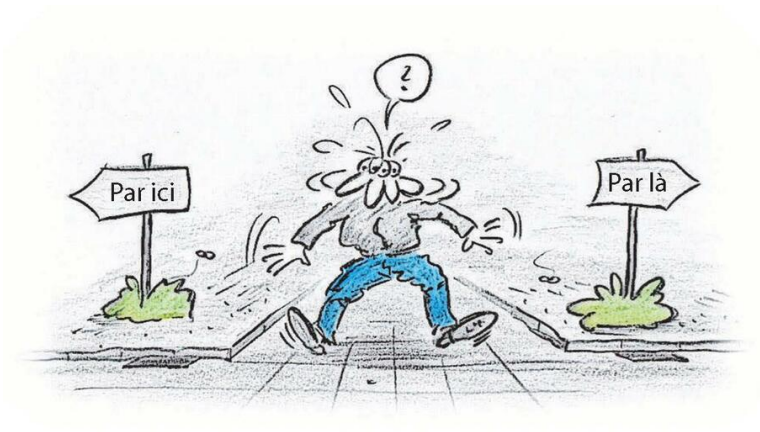
La première ligne d'une fonction est désignée sous le nom de signature et constitue la ligne de déclaration d'une fonction. La définition d'une fonction se trouve entre accolades. Si une fonction doit retourner quelque chose à l'appelant, le type de données de retour, `float` ici, doit se trouver en tête d'instruction. Cela implique cependant la présence obligatoire d'une instruction `return` dans la définition, celle-ci assurant au final le retour d'une valeur, ce qui permet de quitter la fonction après son appel. On peut transmettre à une fonction aucune, une ou plusieurs valeurs à son appel. Dans le cas présent, deux valeurs de type de données `float` sont attendues, qui doivent être transmises à l'appel de la fonction aux paramètres `a` et `b` indiqués. Ces paramètres fonctionnent comme des variables locales, qui sont retirées de la mémoire une fois que la fonction est terminée, car elles ne sont plus nécessaires. Les deux fonctions prépondérantes dans l'environnement de développement Arduino sont naturellement les fonctions `setup` et `loop`, qui sont indispensables :

```
void setup() {/* ... */}
void loop() {/* ... */}
```

Vous voyez que les deux disposent du type de données de retour `void`, dont la traduction est *vide*, parce qu'elles ne fournissent pas de valeur en retour. Vous ne voyez pas non plus de liste de paramètres, ce qui se caractérise par un vide entre les parenthèses. Il est également possible de ne pas transmettre de valeurs à l'appel de la fonction. Il y aurait bien sûr beaucoup d'autres choses à dire sur les fonctions, mais cela est un sujet pour d'autres montages ou parties de tutoriels sur le langage de programmation C++ et serait trop fastidieux à expliquer dans ce livre.

Que sont les structures de contrôle ?

Les instructions ont déjà été abordées au chapitre 2. Elles informent le microcontrôleur de ce qu'il doit faire. Mais un sketch se compose généralement de toute une série d'instructions qui doivent être traitées de manière séquentielle. La carte Arduino présente un certain nombre d'entrées ou de sorties auxquelles vous pouvez raccorder divers composants électriques ou électroniques. Si le microcontrôleur est censé réagir à certaines influences extérieures, vous branchez par exemple un capteur sur une entrée. La forme de capteur la plus simple est un commutateur ou un bouton-poussoir. Quand le contact est fermé, une LED est censée s'allumer. Le sketch doit donc être en mesure de prendre une décision. Si le commutateur est fermé, la LED est alimentée (LED allumée) ; si le commutateur est ouvert, la LED est privée d'alimentation (LED éteinte).



QU'EST-CE QU'UNE STRUCTURE DE CONTRÔLE ?

Les structures de contrôle sont des instructions dans les langages de programmation impératifs. Elles sont utilisées pour commander le déroulement d'un programme et le dévier dans une certaine direction. Une structure de contrôle peut ainsi être une déviation ou une boucle. Son exécution est la plupart du temps commandée par des expressions logiques (booléennes).

Jetons d'abord un coup d'œil sur l'organigramme qui nous montre comment se déroule l'exécution du sketch sur certains circuits, dans la mesure où il ne s'agit plus d'un parcours linéaire. Le sketch se trouve à la croisée des chemins quand une structure de contrôle est atteinte et doit voir ce qu'il est censé faire. Il doit évaluer une condition pour prendre une décision.

L'instruction `if-then` *(si-alors)*

Techniquement, nous utilisons l'instruction `if-then` (*si-alors*). Il s'agit d'une *décision si alors*.

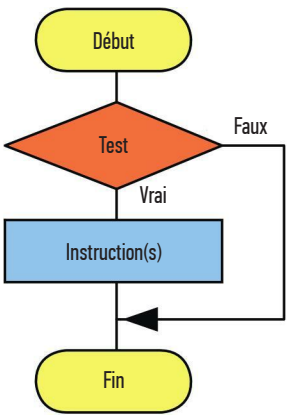


Figure 3
Organigramme d'une structure de contrôle si-alors

Si la condition a été vérifiée, il s'ensuit l'exécution d'une, voire de plusieurs instructions. Voici encore un court exemple :

```
if(boutonStatut == HIGH)
 digitalWrite(ledPin, HIGH);
```

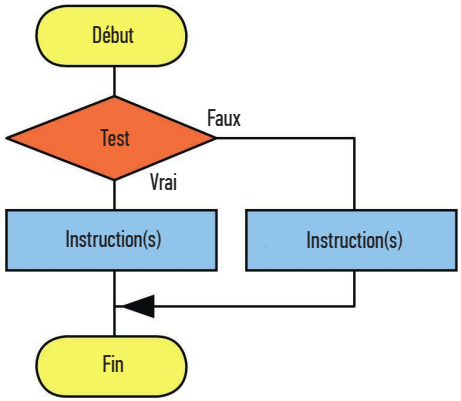
Si plusieurs instructions sont exécutées dans une instruction `if`, vous devez constituer un bloc d'instructions avec les paires d'accolades. Il sera alors exécuté en tant qu'unité d'instructions complète :

```
if(boutonStatut == HIGH)
{
 digitalWrite(ledPin, HIGH);
 Serial.println("Niveau HIGH atteint.");
}
```

L'instruction `if-else` *(si-alors-sinon)*

Il existe aussi une forme élargie de la structure de contrôle si-alors. Il s'agit d'une *décision si-alors-sinon* qui résulte d'une instruction `if-else`. La figure suivante présente l'organigramme :

Figure 4 ▶
Organigramme d'une structure de contrôle si-alors-sinon



L'exemple de code suivant vous montre la syntaxe de l'instruction `if-else`.

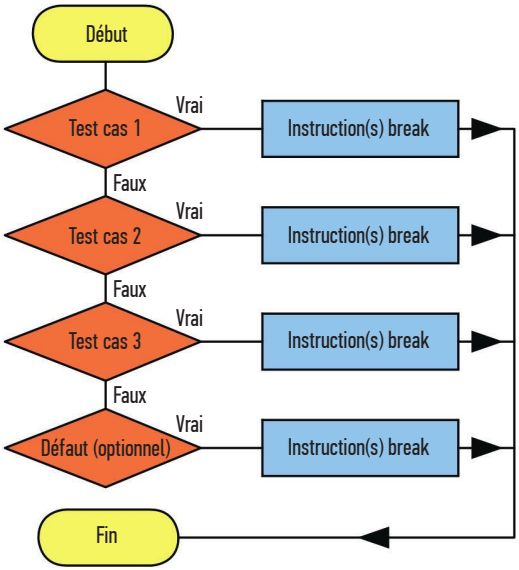
```
if(boutonStatut == HIGH)
 digitalWrite(ledPin, HIGH);
else
 digitalWrite(ledPin, LOW);
```

L'instruction conditionnelle (switch-case)

Lorsque plusieurs interrogations se succèdent, il est bien évidemment possible de donner plusieurs instructions `if-then-else` à l'aide d'une construction. Il existe cependant une version plus facile, avec une écriture simplifiée qui est ainsi plus lisible : l'instruction `switch-case` (figure 5).

La syntaxe se présente comme suit :

```
switch(var)
 { case label1:
// Instruction(s)
 break;
case label2:
// Instruction(s)
 break;
default:
// Instruction(s) break;
}
```



◀ **Figure 5**
Organigramme d'une structure switch-case

Les paramètres suivants sont autorisés :

- `var` : une variable avec les types de données autorisés `int` et `char` ;
- `label1`, `label2` : des constantes avec les types de données autorisés `int` et `char`.

Vous devez impérativement veiller à ce qu'après l'exécution d'une instruction, la boucle soit interrompue avec l'instruction `break`, sinon les marqueurs de saut suivants seront aussi vérifiés et les instructions qu'ils contiennent pourraient être exécutées. Le dernier texte source d'instruction `break` après l'instruction `default` n'est pas obligatoire.

Opérateurs

Bien sûr, dans les structures de contrôle et les conditions à tester, il n'y a pas que la vérification de l'égalité. Le tableau ci-après vous montre tous les opérateurs de comparaison du langage C++ :

Opérateur de comparaison	Signification	Exemple
==	est égal à	`if(a==b) {...}`
<=	est inférieur ou égal à	`if(a<=b) {...}`
>=	est supérieur ou égal à	`if(a>=b) {...}`
<	est inférieur à	`if(<b) {...}`
>	est supérieur à	`if(a>b) {...}`
!=	n'est pas égal à	`if(a!=b) {...}`

◀ **Tableau 2**
Opérateurs de comparaison

Il existe également des liaisons par l'intermédiaire d'opérateurs logiques, qui autorisent plusieurs conditions devant être testées :

Tableau 3 ▶ Opérations logiques

Opération logique	Fonction	Signification	Exemple
!	PAS (NOT)	Inversion de l'état logique. Le résultat est vrai (true) si l'opérande est faux (false).	`if(!a) {...}`
&&	ET (AND)	Le résultat est vrai (true) si les deux opérandes sont vrais.	`if((a<=b)&&(c==5)) {...}`
||	OU (OR)	Le résultat est vrai (true) si l'un des deux opérandes est vrai.	`if((a>=b)||(c<=6)) {...}`

Les boucles

Dans un sketch, l'exécution de plusieurs étapes récurrentes peut s'avérer nécessaire pour calculer des données. Si ces étapes sont similaires, il n'est pas nécessaire de les écrire en grand nombre les unes en dessous des autres et de les faire exécuter de manière séquentielle. Une structure de programme spéciale a été créée à cet effet, permettant d'exécuter une partie de programme composée d'une ou plusieurs étapes, maintes fois répétées. Cette structure s'appelle une *boucle*.

QU'EST-CE QU'UNE BOUCLE ?

Une boucle, appelée *loop* en anglais, est une structure de contrôle dans un langage de programmation. Elle répète un bloc d'instructions défini, appelé le corps de boucle tant que la condition de la boucle logique est vérifiée. Les boucles, dont la condition est toujours vérifiée ou dans lesquelles aucune condition n'a été définie, sont appelées des boucles sans fin.

Voyons comment une boucle est construite. Il existe deux sortes de boucles :

- les boucles avec condition de sortie en tête ;
- les boucles avec condition de sortie en queue.

Ces deux familles de boucles possèdent une *instance* qui contrôle si la boucle doit être itérée, et de quelle manière elle doit l'être. À cette instance est rattachée une seule instruction ou tout un bloc d'instructions (corps de boucle), qui est piloté et traité par l'instance.

Boucles avec condition de sortie en tête

Dans les boucles avec condition de sortie en tête, l'instance de contrôle se situe, comme son nom l'indique, au début de la boucle. L'exécution de la première itération de la boucle dépend de l'évaluation de la condition. Pour les boucles dont la condition se situe en tête, les instructions placées

au sein de la boucle peuvent ne pas être exécutées si la condition est fausse dès l'entrée dans la boucle.

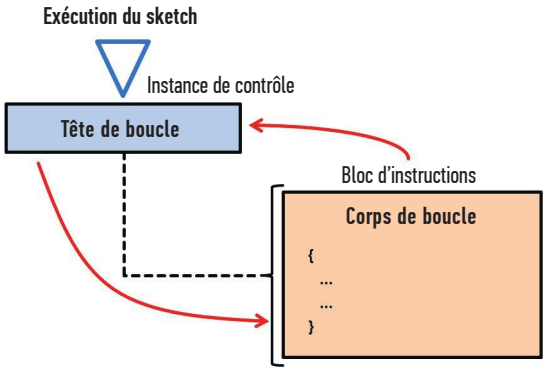


Figure 6
Principe d'une boucle avec condition de sortie en tête

L'utilisation du pluriel dans le titre de cette section indique qu'il existe plusieurs types de boucles avec conditions de sortie en tête, utilisées dans différentes situations.

La boucle `for`

La boucle `for` est toujours utilisée quand on connaît déjà le nombre d'itérations de la boucle avant même de l'appeler. Examinons l'organigramme qui sert de représentation graphique du programme.

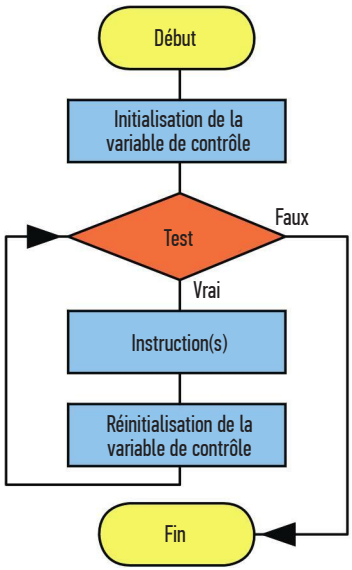


Figure 7
Organigramme d'une boucle for

Une variable appelée *variable de contrôle* est utilisée dans la boucle. Dans la condition, elle est soumise à une évaluation qui décide si et combien de

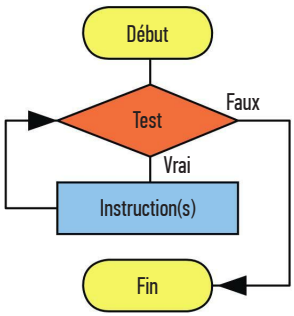
fois la boucle doit être itérée. La valeur de cette variable est généralement modifiée au début de la boucle à chaque nouvelle itération, si bien que la condition d'interruption doit être remplie au bout d'un moment dans la mesure où vous n'avez pas fait de faute de raisonnement. Voici un court exemple, sur lequel nous reviendrons bientôt :

```
for(int i = 0; i < 7; i++)
 pinMode(ledPin[i], OUTPUT);
```

La boucle `while`

La boucle `while` est utilisée quand on ne sait qu'au moment de l'exécution si et combien de fois la boucle doit être itérée. Si, pendant une itération de boucle, une entrée du microcontrôleur par exemple est constamment interrogée ou surveillée alors qu'une action est censée être exécutée à une certaine valeur, cette boucle vous rendra bien service. Voyons maintenant à quoi ressemble l'organigramme :

Figure 8 ▶ Organigramme d'une boucle while



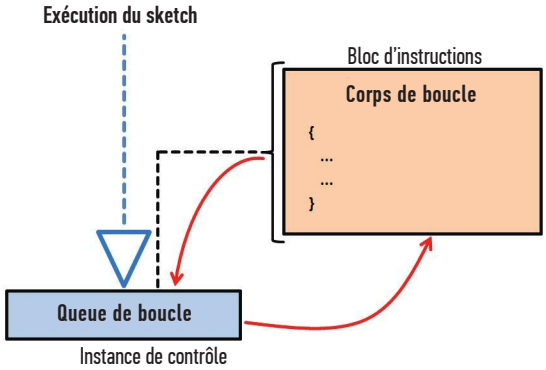
Sur cette boucle, la condition d'interruption se trouve également en tête. En revanche, la variable indiquée dans la condition n'est pas modifiée. Elle doit l'être dans le corps de boucle, faute de quoi nous aurions affaire à une *boucle sans fin* dont nous ne pourrions pas sortir tant que le sketch s'exécute. Voici encore un court exemple d'une boucle `while` :

```
while(i > 1) { // Instance de contrôle
 Serial.println(i);
 i = i - 1;
}
```

Quand vous travaillez avec des valeurs ou des variables du type `float`, par exemple dans l'instance de contrôle, il peut être très risqué d'attendre un résultat précis à cause de l'imprécision de `float`. La condition d'interruption risque de ne jamais être remplie et le sketch se retrouvera prisonnier d'une boucle sans fin. Au lieu de l'opérateur `==` pour évaluer l'égalité, utilisez de préférence les opérateurs `<=` ou `>=`.

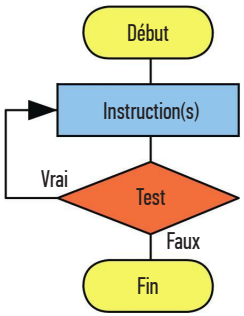
Boucle avec condition de sortie en queue

Venons-en maintenant à la boucle avec condition de sortie en queue. On l'appelle ainsi parce que l'instance de contrôle est hébergée en fin de boucle.



◀ Figure 9
Principe d'une boucle avec condition de sortie en queue

On l'appelle boucle `do... while`. La condition étant évaluée seulement à la fin de la boucle, on peut déjà en déduire qu'elle sera exécutée au moins une fois.



◀ Figure 10
Organigramme d'une boucle do-while

Cette boucle est peu utilisée, mais je tenais tout de même à la citer par souci d'exhaustivité. La syntaxe ressemble à celle de la boucle `while`, mais vous remarquez que l'instance de contrôle est placée en fin de boucle.

```
do {
 Serial.println(i);
 i = i - 1;
} while(i > 1); // Instance de contrôle
```

Vous utiliserez les différentes formes de boucle lorsque vous écrirez vos sketches.

Commentez vos codes !

Quand on traite un problème en programmeur et que l'on code, il peut être utile de prendre çà et là quelques notes. Il nous vient parfois une fulgurance ou une idée géniale et, quelques jours plus tard, nous avons du mal, et c'est souvent le cas pour moi, à nous souvenir exactement de notre raisonnement. Qu'est-ce que j'ai bien pu programmer et pourquoi m'y suis-je pris ainsi et pas autrement ? Chaque programmeur peut bien sûr avoir sa propre stratégie de prise de notes : bloc-notes, dos de prospectus publicitaires, documents Word, etc. Toutes ces méthodes présentent cependant des inconvénients non négligeables.

- Où ai-je bien pu mettre mes notes ?
- S'agit de la dernière version actualisée ?
- Je n'arrive pas à me relire !
- Comment remettre ces notes à un ami qui s'intéresse également à ma programmation ?

Le problème vient de la séparation du code de programmation et des notes, qui ne forment alors plus un tout. Si les notes sont perdues, vous aurez vraiment beaucoup de mal à tout reconstruire. Imaginez maintenant votre ami, qui n'a absolument aucune idée de ce que vous vouliez faire avec votre code. Mais il existe une autre solution : vous pouvez laisser des remarques et consignes dans le code, et ce, à l'endroit précis où elles sont pertinentes. Vous avez ainsi sous la main toutes les informations qui vous sont nécessaires.

Commentaires sur une ligne

Voici un exemple pris dans un programme :

```
int ledBrocheRougeAuto = 7;
// La broche 7 commande la LED rouge (feu tricolore) int
ledBrocheJauneAuto = 6;
// La broche 6 commande la LED jaune (feu tricolore) int
ledBrocheVertAuto = 5;
// La broche 5 commande la LED verte (feu tricolore)
...
```

Ici, des variables sont déclarées et initialisées avec une valeur. Des noms évocateurs ont certes été choisis, mais je trouve utile de laisser encore quelques brèves remarques complémentaires. Derrière la ligne d'instruction est ajouté un commentaire, introduit par deux barres obliques (slash). Pourquoi ces barres sont-elles nécessaires ? C'est simple. Le compilateur essaie bien entendu d'interpréter et d'exécuter tous les prétendus ordres qui lui sont donnés. Prenons par exemple le premier commentaire :

```
La broche 7 commande la LED rouge (feu tricolore)
```

Il s'agit des divers éléments d'une phrase que le compilateur ne comprend pas puisque ce ne sont pas des instructions. Cette notation entraînerait une erreur pendant la compilation du code. Mais les deux `//` masquent cette ligne et informent le compilateur *que tout ce qui suit les deux traits obliques ne le concerne pas et qu'il peut sans crainte ne pas en tenir compte.* C'est une sorte de pense-bête pour le programmeur, qui n'est même pas capable de retenir les choses les plus simples au-delà d'une certaine durée (> 10 minutes). Soyez un peu indulgent avec lui ! Ce mode d'écriture permet d'introduire un commentaire d'une seule ligne.

Commentaires sur plusieurs lignes

Si, en revanche, vous voulez écrire plusieurs lignes, comme une description succincte de votre sketch, placer deux barres obliques devant chaque ligne peut s'avérer fastidieux. Aussi la variante à plusieurs lignes suivante a-t-elle été créée.

```
/*
 Auteur: Erik Bartmann
 Scope: Commande feux tricolores
 Date: 10.12.2020 HP:     https://erik-bartmann.de/
*/
```

Ce commentaire présente une combinaison de signes introductifs `/*` et une combinaison de signes conclusifs `*/`. Tout ce qui se trouve entre les deux *tags* (un *tag* étant une marque utilisée pour identifier des données qui ont une importance particulière) est considéré comme étant du commentaire et ignoré par le compilateur. Tous les commentaires s'affichent en gris dans l'environnement de développement Arduino, de façon à être immédiatement reconnaissables.

Voilà donc le clavier sur lequel vous devrez jouer à l'avenir pour écrire des sketches. Dans ce chapitre, je vous ai exposé de manière assez abstraite je l'avoue quelle méthode était utilisée pour expliquer clairement à une machine ce qu'elle doit faire. Vous allez maintenant appliquer ces connaissances dans les différents montages. Vous pourrez ainsi étendre votre savoir grâce à la mise en pratique de ces méthodes. Je vous conseille de relire de temps en temps ce chapitre pour bien vous familiariser avec ces informations.

Chapitre 4

La carte Arduino Discoveryboard

Au cours du chapitre précédent, vous avez appris des connaissances utiles sur la programmation, connaissances qui vous aideront sûrement à l'avenir à réaliser vos idées. J'aimerais vous présenter au cours de ce chapitre une possibilité pratique qui consiste à installer sur une platine des composants électriques et électroniques dont nous aurons toujours besoin et que vous pourrez utiliser plus tard pour vos projets et vos montages Arduino. Cette platine n'est pas obligatoire. Toutefois, elle vous simplifiera grandement la tâche dans certaines situations pour réaliser vos montages électroniques.

Si vous n'avez pas beaucoup d'expérience en matière de soudure, je vous recommande de commencer par les montages les plus simples décrits dans ce livre afin de vous familiariser avec la technique et l'assemblage de composants électriques et électroniques, puis de monter la carte Arduino Discoveryboard. L'effort supplémentaire que vous devrez accomplir au départ vous fera gagner beaucoup de temps par la suite, car vous n'aurez pas besoin d'assembler constamment les mêmes composants.

La figure de la page suivante représente la platine terminée dont je vais vous présenter la construction en détail dans ce montage. Elle réunit différents composants qui sont soudés sur une plaque d'essai, ce qui vous permettra de les avoir tous sous la main. Je l'ai appelée *Arduino Discoveryboard*.

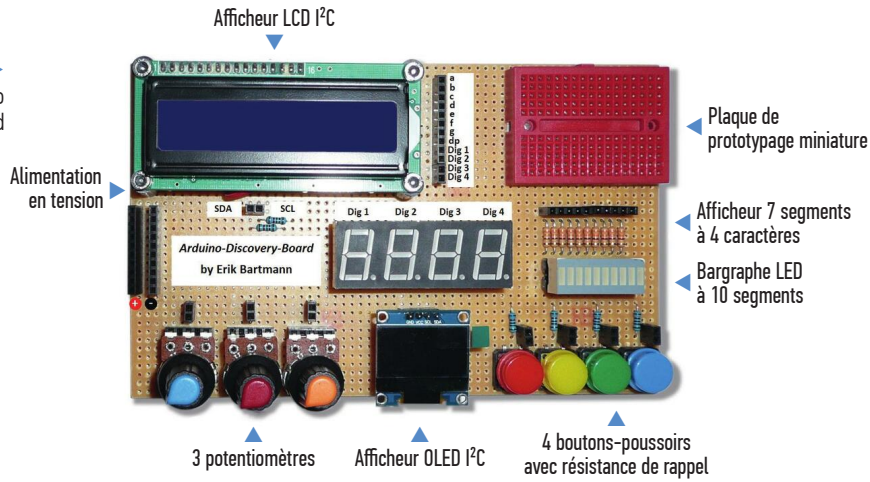


Figure 1 ▶
La carte Arduino Discoveryboard

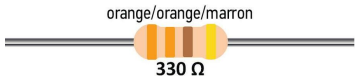
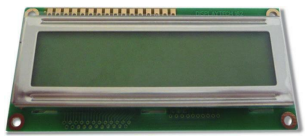
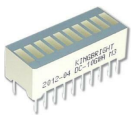
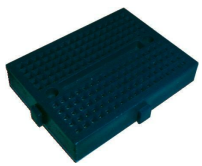
L'idée m'est venue de fabriquer cette carte lorsque j'ai eu besoin pour la nième fois de plusieurs potentiomètres pour réguler les entrées analogiques de la carte Arduino dans le but de réaliser le moniteur analogique. Chaque fois, il fallait que je retourne toute ma réserve de composants pour y retrouver les potentiomètres nécessaires aux diverses expériences afin de les raccorder à l'alimentation, puis de relier les contacts flottants aux entrées analogiques A0 à A5. Cela ne pose pas vraiment de problème en soi. Mais il est fastidieux à la longue de recommencer sans cesse les mêmes opérations. Lorsqu'à cela s'ajoutent des interrupteurs ou des boutons-poussoirs, vous comprendrez à quel point il est préférable que tout soit réuni au même endroit. Seuls prérequis : une main sûre et les composants énumérés ci-après. N'essayez pas forcément de reproduire la Discoveryboard présentée ici. Inspirez-vous-en pour créer votre Discoveryboard personnelle en disposant à votre guise sur une platine les composants que vous utilisez souvent.

Composants nécessaires

Pour la carte Arduino Discoveryboard, nous aurons besoin des composants suivants :

Tableau 1 ▶
Liste des composants

Composant	
3 boutons-poussoirs miniatures (avec bouchon de couleur)	

Composant	
6 résistances de 10 kΩ	
14 résistances de 330 Ω	
3 potentiomètres de 10 kΩ	
1 écran LC-I²C	
1 bargraphe, par ex. Kingbright DC-10EWA	
1 écran OLED 2,4 cm (0,96 pouce), 128 x 64 pixels, pilote LCD	
3 barrettes femelles à 64 broches (RM : 2,54)	
1 afficheur sept segments à 4 caractères, par ex. CL5641BH	
1 plaque de prototypage miniature	

◀ Tableau 1
Liste des composants (suite)

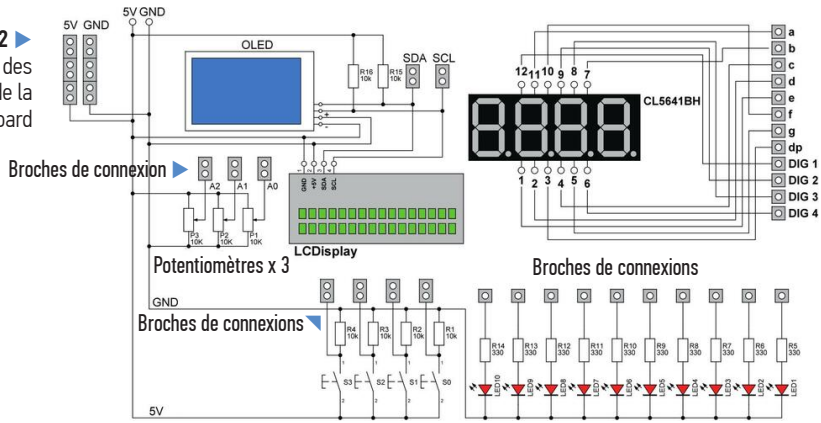
Tableau 1 ▶ Liste des composants (suite)

Composant	
1 platine perforée 160 x 100 (RM : 2,54)	
4 pieds en caoutchouc	

Schéma des connexions

Le schéma des connexions est vraiment simple et facile à comprendre.

Figure 2 ▶ Le schéma des connexions de la Discoveryboard



Pour la soudure au dos de la platine, il faut avoir la main très sûre. Mais si j'y suis arrivé, vous y parviendrez aussi. Comme les points de soudure sont très rapprochés, il est conseillé d'avoir une pompe à dessouder à portée de main, car deux points de soudure très proches risquent de se rejoindre. Ce n'est pas dramatique mais c'est énervant. Cela m'est arrivé très souvent.

L'afficheur sept segments

Nous utilisons pour l'affichage un afficheur sept segments à quatre caractères avec une anode commune, de type CL5641BH. Il est évidemment possible d'utiliser d'autres composants présentant des spécifications similaires. La disposition des broches de l'afficheur est illustrée page suivante.

Je décrirai plus en détail dans le montage n° 12 comment commander cet afficheur sept segments. On peut voir sur le schéma des connexions que tous les segments des différentes positions sont reliés entre eux. Cela signifierait évidemment que tous les segments à toutes les positions s'allumeraient en même temps, ce qui serait complètement stupide. Le point crucial est la commande ciblée ou intelligente des différentes anodes des positions Dig1 à Dig4. Ces anodes sont toutes reliées entre elles, d'où le nom d'*anode commune*. Un procédé particulier entre en application pour commander maintenant chaque position avec des valeurs de positions différentes. Cela devient intéressant !

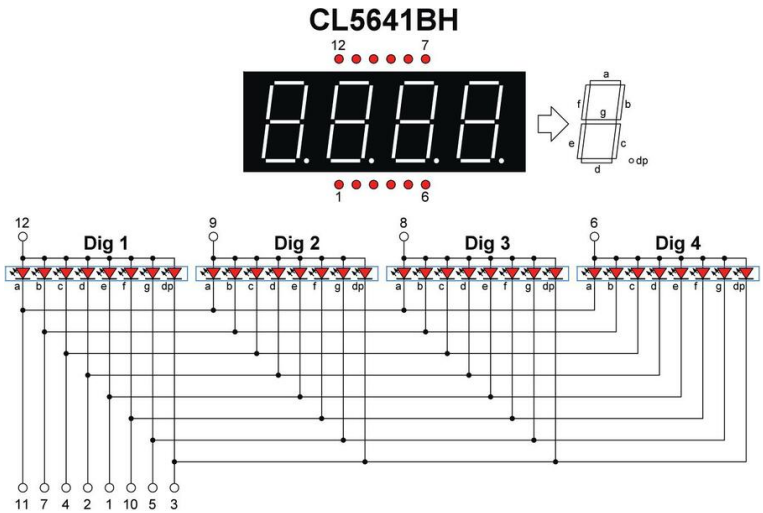


Figure 3
La disposition des broches de l'afficheur CL5641BH

Partie 2

Les montages

Hello World – faire clignoter une LED

La plupart des manuels sur les langages de programmation commencent par la présentation d'un programme appelé *Hello World*. Il donne un premier aperçu de la syntaxe du langage de programmation en affichant le texte Hello World dans une fenêtre. De cette manière, un programmeur peut se faire une idée du langage de programmation et de sa syntaxe, et gagner ainsi du temps.

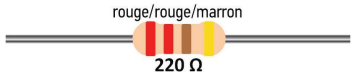
« Hello World » clignote

À quoi ressemble un programme *Hello World* dans l'univers Arduino ? En effet, notre Arduino n'a pas d'écran à l'origine, et donc pas de console de visualisation pour nous informer. Alors que faire ? Si aucune communication sous une forme écrite n'est possible, peut-être l'est-elle avec des signaux optiques ou acoustiques ? Nous optons pour la variante optique, car une diode électroluminescente, également appelée *LED*, se branche sans problème à l'une des sorties numériques et attirera à coup sûr l'attention. J'ai été moi-même très étonné quand ça a marché du premier coup. Voyons d'abord la liste des composants.

Composants nécessaires

Ce montage ne nécessite pas grand-chose et il peut même se passer de composants supplémentaires. En effet, la carte Arduino comporte une LED qui est désignée par la lettre *L*. Toutefois, j'aimerais compléter ce montage par quelques composants que nous réutiliserons pour d'autres montages.

Tableau 1-1 ▶
Liste des composants

Composant	
1 LED rouge	
1 résistance de 220 Ω	

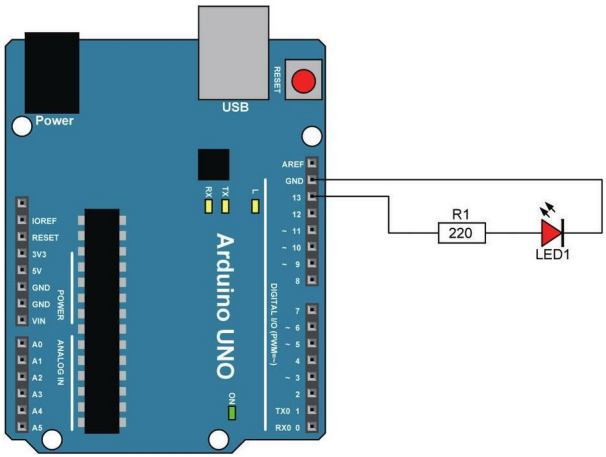
Avant d'examiner le programme Arduino, ou sketch, jetons d'abord un œil au schéma. Ce schéma représente le circuit du montage sous forme graphique, certains l'appellent aussi le schéma des connexions.

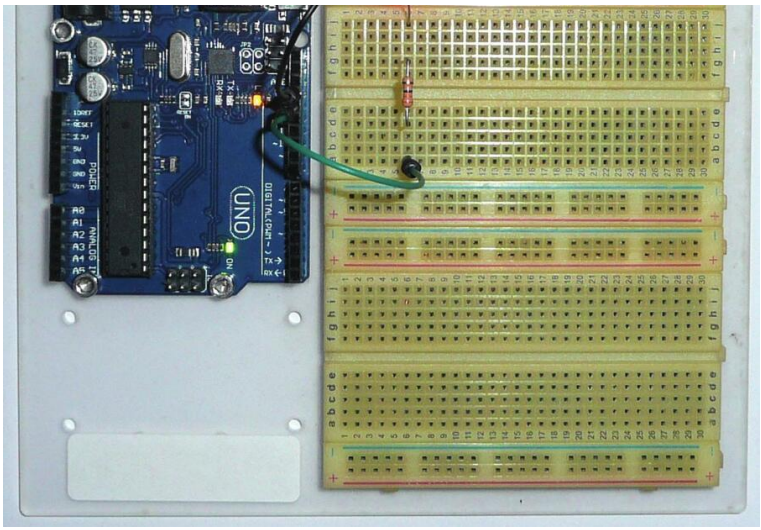
Le schéma des connexions

Le schéma ne représente qu'une seule LED avec la résistance correspondante.

L'anode de la LED (la tige longue) est reliée à la broche 13 via la résistance série de 220 Ω, tandis que l'autre extrémité, ou cathode (la tige courte), de la LED est reliée à la masse (GND) de la carte Arduino. Pour ce circuit, j'utilise pour la première fois la carte Arduino combinée à une plaque de prototypage, dont je vous ai déjà parlé dans le chapitre d'introduction (chapitre 2). J'utiliserai dans les montages suivants l'Arduino Discoveryboard, mais vous pouvez évidemment réaliser de votre côté tous les montages sur une plaque de prototypage tout à fait normale.

Figure 1-1 ▶
Le schéma des connexions





◀ **Figure 1-2**
Le circuit sur une petite carte combinée Arduino

ATTENTION À LA POLARITÉ DE LA LED

Veillez à respecter la polarité correcte de la LED, faute de quoi nous ne verrons qu'une LED éteinte. Vous ne risquez pas d'endommager quoi que ce soit avec une LED mal raccordée mais, tant qu'à faire les choses, faisons-les correctement.

Le sketch Arduino

Le code de notre premier sketch a pour effet de faire clignoter toutes les secondes une LED raccordée à une résistance. Le programme est le suivant :

```
int ledPin = 13;                // Variable déclarée
                                // + initialisée avec broche 13
void setup() {

  pinMode(ledPin, OUTPUT);      // Broche numérique 13
                                // définie comme sortie
}

void loop() {
  digitalWrite(ledPin, HIGH);  // LED au niveau HIGH (5 V)
  delay(1000);                 // Attendre une seconde (1000 ms)
  digitalWrite(ledPin, LOW);   // LED au niveau LOW (0 V)
  delay(1000);                 // Attendre une seconde (1000 ms)
}
```

Je recommande vivement de taper vous-même le code. Je trouve que cela est bénéfique sur le plan pédagogique. Vous apprendrez par exemple à

toujours mettre un point-virgule à la fin d'une ligne d'instruction. Vous vous familiariserez plus avec les finesses de la programmation si vous entrez vous-même les lignes de code. Cela s'applique en particulier au code du premier montage. Pour les montages suivants, le code est parfois si long que le taper à la main serait trop fastidieux.

Le code utilisé dans ce livre est disponible à l'adresse :

https://www.editions-eyrolles.com/dl/0100583

LA STRUCTURE DE BASE D'UN SKETCH ARDUINO

Tous les sketches Arduino présentent la même structure. Celle-ci est représentée sur la figure 3 :

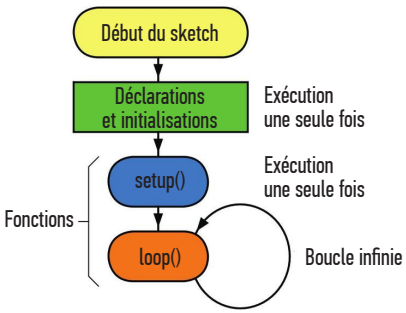


Figure 1-3 ▶ La structure de base d'un sketch

L'ensemble du programme, après le démarrage du sketch, est divisé en trois blocs, comme vous pouvez le voir ci-dessus. Chaque bloc a une fonction clairement définie dans un sketch.

Bloc vert : déclarations et initialisations

Dans ce premier bloc, des bibliothèques externes sont par exemple intégrées, si nécessaire, à l'aide de l'instruction `#include`. Vous apprendrez dans les montages suivants comment cela fonctionne. C'est également l'endroit idéal pour déclarer des variables globales qui seront visibles et utilisables dans l'ensemble du sketch. La déclaration permet de définir à quel type de données la variable doit être affectée. L'initialisation en revanche affecte une valeur à la variable.

Bloc bleu : fonction setup()

Les différentes broches du microcontrôleur sont programmées la plupart du temps dans la fonction `setup()`. Celle-ci définit donc quelles broches doivent servir d'entrées et de sorties. Certaines broches sont connectées à des capteurs comme des interrupteurs ou des résistances sensibles à la température, qui dirigent des signaux de l'extérieur vers une entrée correspondante. D'autres broches transmettent les signaux aux sorties, pour commander par exemple un moteur ou une diode lumineuse.

Bloc orange 3 : fonction loop()

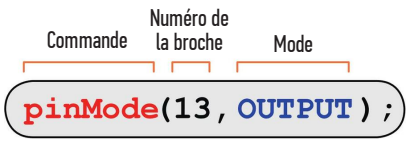
La fonction `loop()` est une boucle sans fin. Elle contient la logique, par exemple lorsque des capteurs doivent être interrogés ou des moteurs commandés en continu. Les deux fonctions, `setup()` et `loop()` constituent ensemble un bloc d'exécution, reconnaissable dans le code aux paires d'accolades {}. Elles servent de limitateurs permettant de savoir où la définition de la fonction commence et où elle se termine.

Les deux fonctions `setup()` et `loop()` doivent avoir ces dénominations exactes, car elles sont recherchées au démarrage du sketch, parce qu'elles servent de points d'entrée pour garantir un démarrage défini. Comment le compilateur pourrait savoir quelle fonction doit être exécutée une seule fois et quelle fonction en continu dans une boucle sans fin ? Ces noms sont donc indispensables à chaque sketch. Ils sont toujours précédés du mot-clé `void`. Il indique que la fonction ne donnera probablement aucune information en retour à la fonction qui l'a appelée.

Revue de code

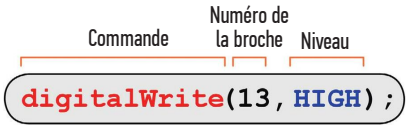
Au départ, nous déclarons et initialisons une variable globale portant le nom de `ledPin` et nous lui attribuons la valeur `13`. À l'aide de l'instruction `int` (int = Integer), nous décidons qu'il s'agit d'un type de donnée entier. Cette variable devient visible dans toutes les fonctions et est accessible. L'initialisation s'apparente à une affectation de valeurs avec l'opérateur d'affectation `=`. La déclaration et l'initialisation s'effectuent ici sur une seule ligne.

La fonction `setup()` est appelée une fois au début du démarrage du sketch et la broche numérique 13 est programmée comme sortie. Examinons encore une fois l'instruction `pinMode`.



Elle présente deux arguments numériques. Le premier pour la broche à configurer et le deuxième pour définir son comportement comme entrée ou sortie. Comme nous voulons raccorder une LED, nous avons besoin pour cela d'une broche de sortie. Le sens de circulation du deuxième argument est indiqué par une constante prédéfinie. Derrière `OUTPUT` se cache la valeur `1`.

Il en va de même pour l'instruction `digitalWrite`, qui présente également deux arguments.



On trouve ici aussi une constante dont le nom est `HIGH`, censée servir d'argument pour un niveau HIGH sur la broche 13. Elle est équivalente à la valeur numérique `1`. Vous trouverez les valeurs correspondantes dans le tableau ci-dessous :

Tableau 1-2 ▶ Les constantes et leurs valeurs

Constante	Valeur	Explication
`INPUT`	`0`	Constante pour l'instruction `pinMode` (programme la broche en tant qu'entrée)
`OUTPUT`	`1`	Constante pour l'instruction `pinMode` (programme la broche en tant que sortie)
`LOW`	`0`	Constante pour l'instruction `digitalWrite` (met la broche au niveau LOW)
`HIGH`	`1`	Constante pour l'instruction `digitalWrite` (met la broche au niveau HIGH)

L'instruction `delay` sert à la temporisation dans le sketch. Elle interrompt l'exécution du sketch pendant un temps correspondant à la valeur donnée qui exprime la durée en millisecondes (ms).

La valeur `1000` signifie une attente de 1 000 ms précisément, soit 1 seconde, avant de continuer.

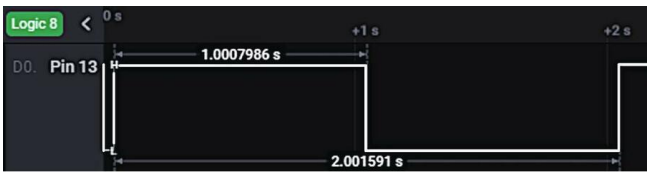
Les différentes étapes de travail de la boucle sans fin, qui a été déclenchée par l'instruction `void loop` sont :

1. allumez la LED de la broche 13 ;
2. attendez 1 seconde ;
3. éteignez la LED de la broche 13 ;
4. attendez 1 seconde ;
5. revenez au point 1.

Illustration de l'évolution temporelle

C'est difficile à voir, mais en regardant bien, on s'aperçoit que la LED sur la carte clignote en même temps que la LED reliée extérieurement. Les LED sont censées commencer à clignoter aussitôt après la transmission réussie sur la carte. L'oscillogramme présente plus précisément l'évolution temporelle de l'impulsion sur la sortie numérique.

J'ai récupéré directement le niveau sur la sortie sans résistance série. Elle alterne constamment entre 0 V et 5 V, ce qui est caractérisé ici par L (LOW) et H (HIGH).



◀ Figure 1-4
Évolution du niveau sur la broche de sortie numérique 13

ATTENTION !

On trouve sur Internet des schémas de circuits où une diode électroluminescente est branchée directement entre masse et broche 13. Les deux fiches femelles se trouvant l'une à côté de l'autre, côté broches numériques, il est très aisé de brancher une LED. Je vous mets expressément en garde contre cette variante, car la LED est utilisée sans résistance série. Ce n'est pas tant pour la LED, mais bel et bien pour votre microcontrôleur que je m'inquiète. J'ai mesuré une fois que l'intensité du courant atteignait 60 mA. Cette valeur est de 50 % au-dessus du maximum et donc assurément trop élevée. Le courant admis par une broche numérique du microcontrôleur est de 40 mA au maximum.

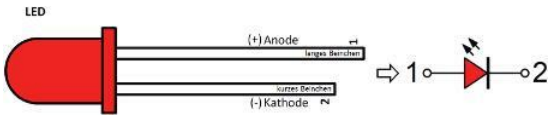
Problèmes courants

Les erreurs ont le don de se glisser rapidement même si vous pensez avoir suivi à la lettre les textes et les figures. Je sais par expérience que la recherche d'erreurs dans un programme ou un circuit est parfois fastidieuse. C'est pourquoi j'indique toujours dans chaque montage les sources d'erreurs possibles. Si votre montage ne fait pas ce que vous attendez de lui, passez en revue les points que j'énumère dans cette section.

Si la LED ne s'allume pas, plusieurs raisons peuvent en être la cause.

La LED est-elle mal polarisée ?

Rappelez-vous les deux connexions d'une LED que sont l'anode et la cathode.



La LED est-elle défectueuse ?

La LED a peut-être été grillée par une surtension lors des montages précédents. Testez-la avec une résistance de 220 Ω sur une source d'alimentation de 5 V.

Y a-t-il une erreur de raccordement ?

Vérifiez les fiches de la barrette de raccordement qui sont reliées à la LED ou à la résistance. S'agit-il bien de GND et de la broche 13 ?

Une erreur se serait-elle glissée dans le code ?

Vérifiez le sketch que vous avez entré dans l'éditeur de l'IDE. Peut-être avez-vous oublié une ligne ou commis une erreur, ou peut-être le sketch a-t-il mal été transmis ? Avez-vous bien mis les accolades au début et à la fin des fonctions `setup()` et `loop()` ? Avez-vous toujours terminé une ligne d'instructions avec un point-virgule ? Ce sont les erreurs les plus fréquentes commises par les débutants.

Si la LED qui se trouve sur la carte clignote, la LED branchée doit elle aussi clignoter. Le sketch fonctionne dans ce cas correctement.

BASES DE CALCULS DE LA RÉSISTANCE SÉRIE

Dans ce montage, j'ai utilisé une résistance de 220 Ω qui est amplement suffisante pour ce petit circuit. Mais vous avez parfaitement le droit de vous interroger sur les raisons qui ont motivé ce choix. Le circuit suivant comporte une diode et une résistance série qui sont branchées à une source d'alimentation.

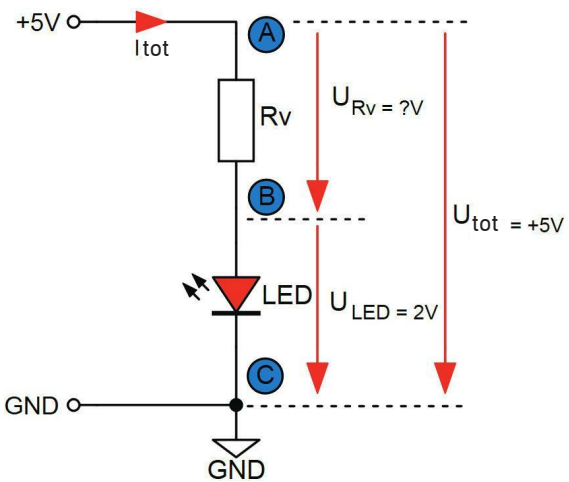


Figure 1-5 ▶ Une LED avec résistance série

J'ai ajouté quelques flèches pour les valeurs de tension et le courant total. Pour calculer la valeur d'une résistance, on utilise la loi d'Ohm que nous avons déjà évoquée. Elle décrit le rapport entre la tension, le courant et la résistance. Si nous amenons une tension *U* à un composant, le courant *I* qui le traverse évolue proportionnellement à la tension. Le rapport entre les deux grandeurs, c'est-à-dire la tension et le courant, est donc constant et il définit la résistance électrique *R*. Nous obtenons alors l'équation suivante :

$$\frac{U}{I} = constante = R$$

Si je me donne la peine de mettre les lettres dans l'ordre, j'obtiens l'équation suivante qui me sert à calculer la résistance électrique :

$$R = \frac{U}{I}$$

Un aspect important n'a pas encore été abordé : qu'est-ce qu'une résistance série ? Les électrons circulant dans un conducteur peuvent avoir plus ou moins de mal à le traverser. Sur leur chemin, ils rencontrent en effet des résistances très différentes les unes des autres. On peut établir une classification des matériaux en fonction de leur conductibilité :

- isolants (très haute résistance). Par exemple : la céramique ;
- mauvais conducteurs (haute résistance). Par exemple : le verre ;
- bons conducteurs (faible résistance). Par exemple : le cuivre ;
- très bons conducteurs (supraconductivité à de très basses températures où la résistance électrique tend vers *0*) ;
- semi-conducteurs (la résistance peut être contrôlée). Par exemple : silicium ou germanium.

Il existe deux grandeurs électriques inversement proportionnelles : la résistance *R* et la conductance *G*. Plus la résistance est élevée, plus la conductance est faible et inversement. La relation qui lie une résistance à une conductance est donnée par :

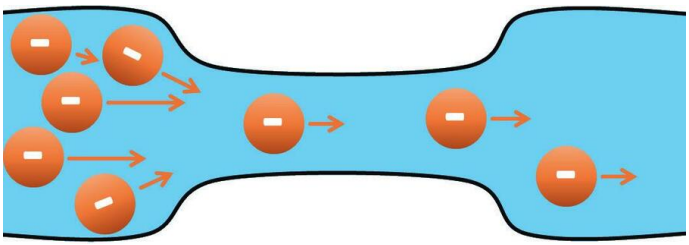
$$R = \frac{1}{G}$$

La résistance est la valeur inverse de la conductance. On peut comparer une résistance élevée à un goulet d'étranglement que doivent franchir les électrons. Le flux de courant est freiné et donc plus faible. Imaginez que vous couriez sur une surface lisse. Avancer ne devrait pas vous poser trop de problèmes. En revanche, si vous essayez de courir dans du sable en gardant la même allure, c'est fatigant. Vous dépensez de l'énergie sous forme de chaleur et votre vitesse diminue. Il en va de même pour des électrons qui doivent traverser par exemple du verre (mauvais conducteur) au lieu du cuivre (conducteur).

Cette résistance n'est évidemment pas sans conséquences. Le frottement plus intense des électrons, par exemple sur la paroi externe ou bien entre eux, dégage une énergie sous forme de chaleur que la résistance transmet à l'extérieur.

La plupart des circuits électroniques recourent à des composants spéciaux, qui réduisent artificiellement le flux de courant. La résistance *R* est alors indiquée en ohms (Ω). Il s'agit de résistances fabriquées spécialement (par exemple des résistances à couche de carbone ou à couche métallique) avec des valeurs différentes, pourvues d'un code couleur qui permet de déduire leur valeur de résistance respective. Nous reviendrons plus loin sur ce point.

Figure 1-6 ▶
Une résistance freinant le flux d'électrons



Lorsque vous examinez de plus près la figure 1-6 avec le flux d'électrons, vous pourriez avoir l'impression que les électrons sur le côté gauche avancent plus vite que ceux du côté droit. Ce n'est pas le cas. Le courant qui circule dans un circuit fermé est toujours le même ! Certes, il est influencé par la résistance illustrée ici, mais le courant avant ou après la résistance est toujours le même. À chaque unité de temps, le même nombre d'électrons passe dans le conducteur ou dans la résistance.

Revenons à notre circuit afin de calculer la résistance. Comment déterminer le courant et la tension ? Ce n'est pas très compliqué. À la résistance série et la LED (entre les points A et C de la figure 1-5), on a +5 V, ce qui correspond à la tension d'alimentation de la carte Arduino. On la retrouve aussi à la sortie de la broche numérique lorsqu'elle est commandée avec un niveau HIGH. Aux bornes de la LED, entre les points B et C, on a, normalement, une chute de tension d'environ 2 V, selon la LED utilisée et sa couleur. La tension aux bornes de la résistance série, donc entre les points A et B, est par conséquent la différence entre +5 V et +2 V, soit +3 V.

Reste à connaître la grandeur du courant qui passe par la résistance. Lorsque les composants sont disposés les uns derrière les autres dans un seul circuit électrique, nous parlons de montage en série. C'est notre cas ici. Dans un montage en série, le courant passe par tous les composants. La fiche technique de la carte Arduino nous apprend que le courant maximum fourni par une broche numérique est de 40 mA. Cette valeur ne doit en aucun cas être dépassée, faute de quoi le microcontrôleur pourrait être endommagé. C'est pourquoi nous limitons le courant à l'aide de la résistance Rv. En électronique, on n'est pas obligé de repousser les limites et on préfère même s'en tenir sagement à l'écart. Pour calculer la résistance série, j'utilise ici deux valeurs de courant différentes soit 5 mA et 10 mA, des valeurs situées entre 5 mA et 30 mA sont courantes pour une LED.

Voyons les calculs correspondants et les résultats obtenus :

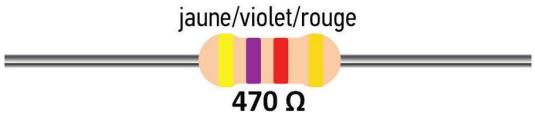
$$R_1 = \frac{U_{ges} - U_{LED}}{I^1} = \frac{5V - 2V}{10\ ma} = 300\ \Omega$$

et

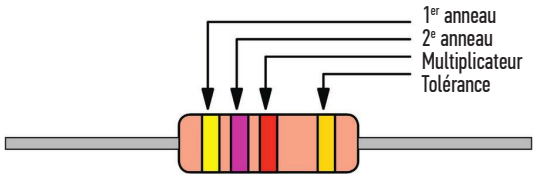
$$R_2 = \frac{U_{ges} - U_{LED}}{I_2} = \frac{5V - 2V}{5\ ma} = 600\ \Omega$$

Pour une telle valeur de résistance, nous pouvons donc employer des valeurs comprises entre 300 Ω et 600 Ω. La sortie de la broche Arduino correspondante ne sera que modérément sollicitée. Une valeur de 330 Ω suffit amplement en ce qui nous concerne. Toutes les valeurs de résistance possibles ne sont pas fabriquées, mais des E-séries normalisées sont proposées avec certaines classes. Il faut également tenir compte de la dissipation d'énergie avec une valeur admise de ¼ watt. Divers assortiments sont disponibles dans le commerce.

Je vous avais promis de vous expliquer la signification des anneaux de couleurs des résistances. La figure ci-dessous présente une résistance dotée de différents anneaux de couleur.



Que signifient-ils exactement et comment déchiffre-t-on le code ? Nous allons l'expliquer en nous servant d'un exemple :



Une résistance possède généralement quatre anneaux de couleur. On a opté pour cette forme de marquage, car il n'y a pas assez de place pour un marquage explicite. Pour interpréter les anneaux de couleur, il faut placer les trois anneaux les plus rapprochés sur le côté gauche. D'après vous, quelle est la valeur de cette résistance ?

1er anneau	2e anneau	3e anneau	4e anneau	Valeur de la résistance
Jaune : valeur 4	Violet : valeur 7	Rouge : multiplicateur 100	Or : +/- 5 %	4,7 kΩ

Quand on écrit ces chiffres les uns à côté des autres, on obtient un résultat parfaitement lisible : 4 700 correspond à 4,7 kΩ. La valeur de tolérance donne

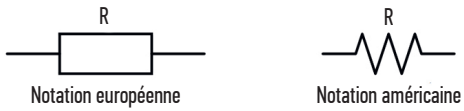
des indications sur la qualité : plus elle est basse, plus la valeur de résistance remplit ses promesses. Mais d'où est-ce que je tiens ces valeurs ? C'est très simple ! Le tableau ci-dessous reprend tous les codes couleurs avec leurs valeurs correspondantes :

Tableau 1-3 ▶ Code couleur des résistances

Couleur	1er anneau	2e anneau	3e anneau	4e anneau
Noir	x	0	$10^0 = 1$	
Marron	1	1	$10^1 = 10$	+/- 1 %
Rouge	2	2	$10^2 = 100$	+/- 2 %
Orange	3	3	$10^3 = 1\,000$	
Jaune	4	4	$10^4 = 10\,000$	
Vert	5	5	$10^5 = 100\,000$	+/- 5 %
Bleu	6	6	$10^6 = 1\,000\,000$	+/- 0,25 %
Violet	7	7	$10^7 = 10\,000\,000$	+/- 0,1 %
Gris	8	8	$10^8 = 100\,000\,000$	+/- 0,05 %
Blanc	9	9	$10^9 =$ 1 000 000 000	
Or	-	-	$10^{-1} = 0,1$	+/- 5 %
Argent	-	-	$10^{-2} = 0,01$	+/- 10 %

Voici comment sont représentées les résistances dans les schémas électriques. Il y a des différences entre les normes européennes et américaines :

Figure 1-7 ▶ Symboles de la résistance fixe dans un schéma électrique



Notation européenne — Notation américaine

Le symbole de l'ohm (Ω) est en principe absent et seul le nombre est indiqué si la valeur est inférieure à 1 kΩ (1 000 ohms), suivi éventuellement d'un *K* pour kilo, si la valeur est supérieure ou égale à 1 kΩ ou d'un *M* pour méga, si la valeur est supérieure ou égale à 1 MΩ. Voici quelques exemples :

Tableau 1-4 ▶ Différentes valeurs de résistance

Valeur	Marquage
220 Ω	220
1000 Ω	1K
4700 Ω	4,7K ou 4K7
2,2 MΩ	2,2M

La dissipation d'énergie maximale admise pour notre montage est de ¼ watt. Il s'agit toujours de résistances à couche de carbone. Elles coûtent moins cher que celles à couche métallique. Vous trouverez des résistances de différentes tailles et couleurs : plus elles sont grandes ou grosses, plus la dissipation d'énergie est élevée.

Voilà pour votre premier montage réalisé avec la carte Arduino ! Vous avez raccordé une LED et une résistance appropriée à la carte Arduino puis, à l'aide du sketch, vous avez fait clignoter cette LED à une cadence que vous avez définie. Vous avez réussi à faire clignoter la LED ? Si la réponse est oui, vous pouvez être fier de vous ! Si vous vous en sentez capable, vous pouvez maintenant vous pencher sur le thème suivant. La MLI est un sujet que vous rencontrerez immanquablement durant votre carrière de développeur amateur. Si vous ne comprenez pas tout tout de suite, n'hésitez pas à revenir sur le sujet plus tard.

LA COMMANDE MLI

Nous en arrivons à la commande d'une LED par MLI. Nous avons déjà vu brièvement de quoi il s'agit au chapitre 1. Il est maintenant temps de passer à la pratique. Nous voulons allumer progressivement une LED à l'aide d'une commande analogique, puis l'éteindre brusquement lorsqu'une valeur maximale est atteinte. Ensuite, tout doit recommencer au début. Il s'agit presque ici d'un clignotement en douceur, contrairement au clignotement brusque du circuit précédent. La commande s'effectue à l'aide des broches numériques dont le numéro est précédé d'un tilde. Il s'agit des broches D3, D5, D6, D9, D10 et D11. Dans cet exemple, j'ai utilisé la broche 3. Le code du sketch est le suivant :

```
int ledPin = 3; // Variable déclarée + initialisée avec broche 3
int pwmValue = 0; // Variable déclarée + initialisée pour MLI

void setup() {/* Aucun code requis */ } void
loop() {

 analogWrite(ledPin, pwmValue++);  // Commande de la LED
                                   // avec valeur MLI
 delay(10);// Courte pause
 if(pwmValue > 255) pwmValue = 0;  // Quand valeur MLI > 255
                                   // -> remettre à 0
}
```

Revue de code

Au départ, nous déclarons et initialisons deux variables globales portant le nom **ledPin** et **pwmValue** et dont le type de données entier est **int** (int = Integer). Examinons l'instruction **analogWrite**.

Elle présente deux arguments : le premier définit la broche et le deuxième la valeur MLI qui peut varier entre 0 et 255.

```
pwmValue++
```

augmente la variable **pwmValue** de la valeur 1 comme indiqué par les deux signes +. En programmation, on parle d'*incrémentation*. Dans un circuit conventionnel, on obtiendrait le même comportement par l'ajout d'une ligne de code supplémentaire :

```
pwmValue = pwmValue + 1;
```

Si on laissait l'incrémentation de la variable **pwmValue** se poursuivre sans surveillance dans une boucle sans fin **loop**, on aboutirait tôt ou tard à une valeur supérieure à 255. Pour l'éviter, l'instruction **if** vérifie si la valeur est supérieure à 255. Si la réponse est positive, l'instruction suivante est exécutée :

```
pwmValue = 0;
```

Cela nous ramène à la valeur initiale de 0. Par conséquent, la LED s'éteint à la prochaine activation de l'instruction **analogWrite**. Nous pourrions aussi modifier le code de telle sorte que la LED ne s'éteigne pas d'un coup, mais que sa luminosité décroisse progressivement.

IMPORTANT !

Pour commander une broche numérique par MLI avec l'instruction **analogWrite**, il n'est pas nécessaire de la programmer au préalable à l'aide de l'instruction **pinMode**. La fonction **setup** ne contient pas d'instruction en ce sens. D'ailleurs, elle ne contient aucun code.

Vous trouverez de plus amples informations sur la MLI aux adresses suivantes :

https://www.arduino.cc/en/Tutorial/PWM

https://www.arduino.cc/en/Reference/AnalogWrite

Bon à savoir

Nous avons vu comment faire clignoter une LED avec quelques lignes de code. On pourrait faire encore plus court.

```
int ledPin = 13; // Variable déclarée + initialisée avec broche 13
void setup() {
  pinMode(ledPin, OUTPUT); // Broche numérique 13 définie comme sortie
}
void loop() {
  digitalWrite(ledPin, !digitalRead(ledPin)); // Bascule la LED

  delay(1000);   // Attendre une seconde
}
```

La ligne décisive est :

```
digitalWrite(ledPin, !digitalRead(ledPin));
```

Nous utilisons ici deux nouvelles structures : il s'agit d'une part de l'instruction `digitalRead` qui permet de lire l'état d'une broche numérique.

D'autre part, nous utilisons *l'opérateur NOT* avec le point d'exclamation (`!`) qui sert à inverser le résultat. Dans le langage de programmation C/C++, il n'existe pas de type de donnée pour les valeurs logiques, comme vrai ou faux. On utilise donc une valeur de type `int`. Dans le tableau 1-2, nous avons vu que le niveau LOW équivaut à la valeur `0` et le niveau HIGH à la valeur `1`. L'opérateur NOT permet de passer de l'un à l'autre, c'est-à-dire de *basculer* entre l'un et l'autre. Vous trouverez de plus amples informations sur les opérateurs booléens à l'adresse suivante :

https://www.arduino.cc/en/Reference/Boolean

Si vous souhaitez approfondir la thématique autour des résistances, vous pouvez lire l'ouvrage *L'électronique par la pratique* de Charles Platt (Éditions Eyrolles).

Qu'avez-vous appris ?

Vous avez appris à déclarer et à initialiser correctement une variable globale.

- Vous connaissez désormais la structure de base du sketch.
- Vous avez déterminé le sens de transmission des données pour une certaine broche comme OUTPUT avec l'instruction `pinMode`, si bien que vous avez pu envoyer un signal numérique (HIGH ou LOW) au moyen de l'instruction `digitalWrite` à la sortie, à laquelle est raccordée la LED.

- Vous avez créé un temps d'attente dans l'exécution du sketch au moyen de l'instruction `delay`, si bien que la LED restait allumée ou éteinte un certain temps.
- Vous avez vu une variante abrégée du code pour faire clignoter la LED à l'aide de l'opérateur *NOT* et de l'instruction `digitalRead`.
- Vous savez que pour utiliser une LED, il faut une résistance série dimensionnée en conséquence.
- Vous savez calculer une résistance série à l'aide de la loi d'Ohm.
- Il est très facile de connaître la valeur d'une résistance indiquée par ses anneaux colorés en se référant au tableau des codes couleurs.
- Vous avez vu comment commander une LED depuis une broche MLI.

Workshop pour faire clignoter une LED

À l'issue de ce montage, voici un petit exercice : modifier le sketch de telle sorte que les temps durant lesquels la LED est allumée ou éteinte soient déterminés par deux variables. Cela vous permet de changer le rapport cyclique. Il correspond au rapport entre la *durée de l'impulsion* et la *durée de la période*. Le résultat est la plupart du temps exprimé en pourcentage. Le chronogramme suivant montre les différentes durées pour *t* ou *T* :

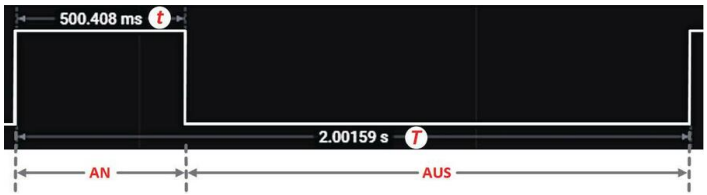


Figure 1-8 ▶ Évolution du niveau sur la broche de sortie numérique 13 à un rapport cyclique de 25 %

t = durée de l'impulsion T = durée de la période

La formule pour calculer le rapport cyclique est la suivante :

$$Rapport\ cyclique = \frac{t}{T}$$

Programmez le sketch de telle sorte que la LED reste allumée pendant 0,5 s et éteinte pendant 1,5 s. Le rapport cyclique se calcule comme suit :

$$Rapport\ cyclique = \frac{t}{T} = \frac{500\ ms}{200\ ms} = 0{,}25 \mathrel{\hat{=}} 25\,\%$$

Ceci correspond à un rapport cyclique de 0,25. Par rapport à la durée complète de la période, la LED est allumée pendant 25 % du temps.

Montage 2

Programmation Arduino de bas niveau

Le cœur de la carte Arduino Uno est son microcontrôleur, qui fait office d'unité centrale traitant les différentes opérations informatiques. Outre l'alimentation, il existe bien sûr d'autres connexions offrant une communication avec le monde extérieur. Il existe ainsi des entrées et sorties analogiques et numériques (nous les avons déjà rencontrées au chapitre 1) qui peuvent être interrogées et utilisées. Comme un microcontrôleur effectue des opérations élémentaires dites de bas niveau à l'aide de signaux numériques, les informations sont stockées sous forme de niveaux HIGH ou LOW (HAUT ou BAS). Certaines d'entre elles peuvent se trouver dans des zones de stockage interne particulières appelées *registres*. Dans ce montage, nous allons raisonner au niveau le plus bas, ce qui peut sembler ardu mais sera très utile pour bien comprendre les bases du système. Mais pas de panique, nous allons procéder pas à pas.

Vous vous demandez certainement ce qu'est un registre. Pour pouvoir être programmé, un microcontrôleur dispose d'un certain nombre d'emplacements de mémoire pour gérer et enregistrer en interne des valeurs. La plupart de ces mémoires présentent un accès en écriture et en lecture, permettant ainsi de stocker et de lire des données. D'autres mémoires peuvent seulement être consultées et ne fournissent qu'un accès en lecture. Ces emplacements de mémoire sont appelés des *registres*. Les ports que nous allons justement découvrir ci-après en font partie.

Les accès du microcontrôleur

Pour communiquer avec un microcontrôleur, vous devez pouvoir envoyer et recevoir des signaux. Pour cette raison, certaines connexions, appelées *ports*,sont proposées. Port vient du latin *porta* qui signifie porte ou accès. Sur la carte Arduino, plusieurs broches de connexions ont été regroupées en blocs fonctionnels. Nommés ports, ces blocs ont une largeur de données de 8 bits. Vous voyez certains de ces ports sur la figure 2-1 :

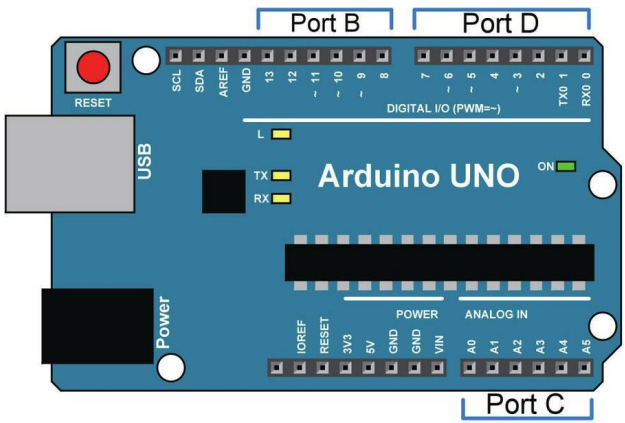


Figure 2-1 ▶
Les ports de la carte Arduino Uno

Les ports signalés ici sont connectés aux broches suivantes :

- port B : broches numériques 8 à 13 ;
- port C : entrées analogiques (A0 à A5) ;
- port D : broches numériques 0 à 7.

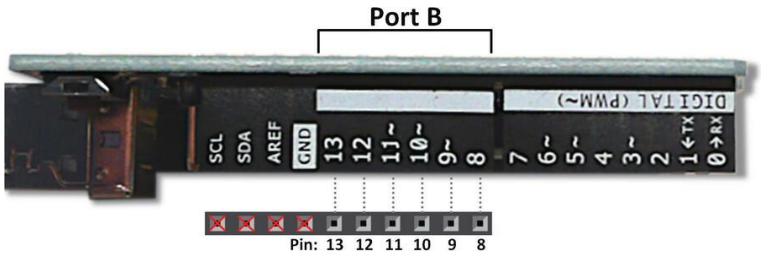
Comme chacun de ces trois ports doit pouvoir être programmé individuellement, des registres supplémentaires sont nécessaires pour les configurer. La petite lettre *x* sera remplacée par le port requis. Ces ports, qui ont également une largeur de données de 8 bits, sont les suivants :

- DDRx *(Data Direction Register)* - Définit une broche comme entrée (0) ou comme sortie (1) : *lecture/écriture* ;
- PORTx *(Pin Output Value)* - Définit l'état de la broche sur HAUT ou BAS : *écriture* ;
- PINx *(Pin Input Value)* - Lit la valeur d'une broche : *lecture seule*.

https://www.arduino.cc/en/Reference/PortManipulation

Programmation d'un port

Regardons cela en détail, même si je précise encore une fois que ce type de programmation est d'un *niveau avancé*, mais qu'il va nous permettre de mieux comprendre tout le système. Jetons un coup d'œil au port B correspondant aux broches numériques 8 à 13.

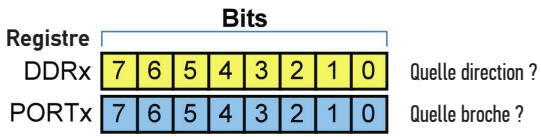


◀ **Figure 2-2**
Port B

Certaines de ces broches, appelées aussi *headers*, ne sont pas affectées à un port. Elles se distinguent des autres broches par leur petite croix. À côté de ces broches figure un libellé qui donne une indication de leur fonction, ce qui est très utile. Les quatre connecteurs de gauche ne peuvent pas être utilisés pour nos projets car ils sont associés au bus I^2C, à l'alimentation en tension et à la masse. Pour pouvoir accéder aux broches numériques 8 à 13, il faut avoir configuré les registres appropriés. Mais il faut d'abord savoir quelles broches doivent fonctionner comme entrées et comme sorties. En résumé, nous avons :

- broches 8, 9 et 10 : entrées ;
- broches 11, 12 et 13 : sorties.

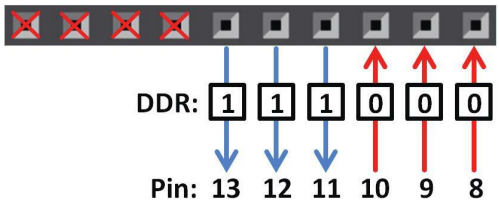
J'ai déjà évoqué les registres, utilisés pour la configuration et le contrôle des ports. La figure 2-3 montre deux registres, ayant une largeur de données de 8 bits, qui assurent ces fonctions. Plus loin dans ce montage, nous ferons connaissance d'un registre supplémentaire.



◀ **Figure 2-3**
Les registres DDRx et PORTx

Ils se nomment *DDRx* et *PORTx*. Commençons par DDRx, qui gère la direction du flux de données, où x est le port. Dans notre cas, ce sera donc DDRB. Nous savons que les entrées sont configurées avec la valeur 0 et les sorties avec la valeur 1.

Figure 2-4 ▶
La configuration du port B



Les flèches indiquent la direction du flux de données et nous devons effectuer la programmation dans l'environnement de développement selon le sketch suivant. Je vous renvoie à mes explications détaillées du montage n° 1 : la fonction de configuration `setup` est exécutée une seule fois au début du sketch et la fonction `loop` est une boucle sans fin.

```
void setup() {
  DDRB = 0b11111000;    // Broches 8, 9, 10 en ENTRÉE. Broches 11,
                        // 12, 13 en SORTIE
  PORTB = 0b00111000;   // Broches 11, 12, 13 au niveau HAUT
}

void loop() {/* vide */}
```

Dans cet exemple, le code n'est présent que dans la fonction `setup` car nous avons juste besoin d'une exécution unique. La fonction `loop` est donc vide. L'attribution d'une valeur au registre peut se faire en format binaire, ce qui simplifie beaucoup la compréhension, car cela montre immédiatement comment chaque broche du port fonctionne. À la suite du préfixe `0b` se situe la valeur binaire :

```
  DDRB = 0b11111000; // Correspond à la valeur décimale 248
```

Composants nécessaires

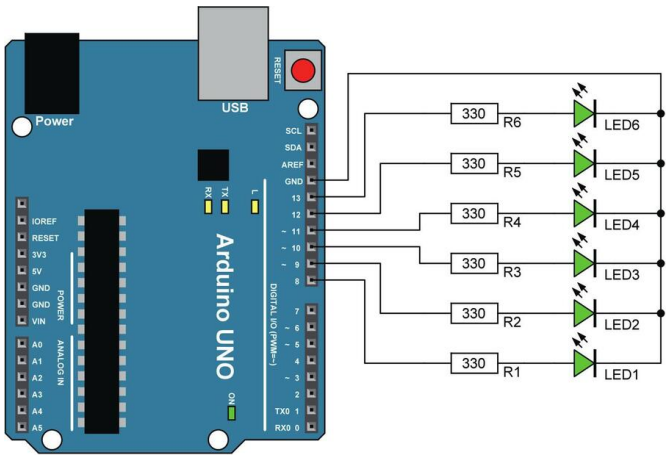
Pour ce montage, nous aurons besoin des composants suivants :

Tableau 2-1 ▶
Liste des composants

Composant	
6 LED vertes	
6 résistances de 330 Ω	

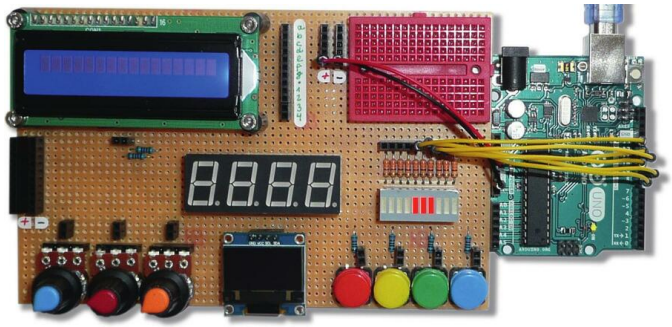
Composant	
3 résistances de 10 kΩ	
3 boutons-poussoirs miniatures	

Le schéma des connexions pour commander les LED est le suivant :



◀ **Figure 2-5**
Le schéma des connexions avec six LED

La réalisation du montage sur une carte Arduino Discoveryboard peut être la suivante :

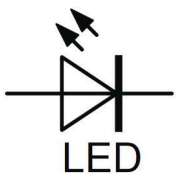


◀ **Figure 2-6**
Configuration du test avec six diodes sur la carte Arduino Discoveryboard

Avec ce montage, il est possible de modifier les différentes valeurs de contrôle pour un résultat immédiatement visible. Vous remarquerez que

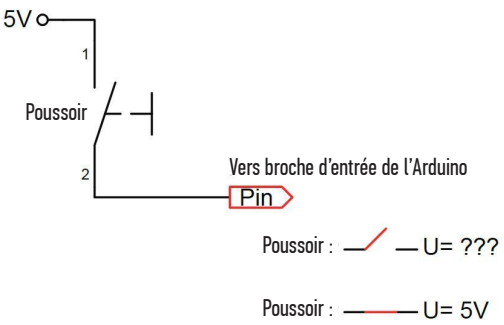
seules les broches 11, 12 et 13 peuvent être configurées comme sorties. Cependant, le montage utilise six diodes reliées aux broches numériques 8 à 13. Que faut-il changer pour que les six diodes électroluminescentes puissent être contrôlées ? La réponse est très facile ! Mais avant cela, regardons de plus près ce qu'est une diode électroluminescente et comment la contrôler. Une diode électroluminescente (appelée aussi LED) est un dispositif semi-conducteur qui émet une lumière d'une certaine longueur d'onde dépendant du matériau semi-conducteur utilisé. Comme le laisse entendre le mot *diode*, le sens de branchement de la diode a une importance, car la LED ne transmet la lumière que dans un sens. Si la LED est branchée à l'envers, elle reste éteinte, ce qui ne veut pas dire qu'elle est défectueuse. Par ailleurs, il est impératif de veiller à ce qu'elle soit *toujours* reliée à une résistance série d'une valeur appropriée. Sinon, elle s'allumera une fois avec une très forte luminosité, mais plus jamais ensuite. Vous avez vu au montage n° 1 comment définir la valeur de la résistance série. 330 Ω est une bonne valeur. Tout comme une diode classique, une LED possède deux contacts, l'*anode* et la *cathode*. Son symbole est quasiment le même qu'une diode normale, avec en plus deux petites flèches représentant le flux lumineux émis :

Figure 2-7 ▶
Les symboles d'une LED

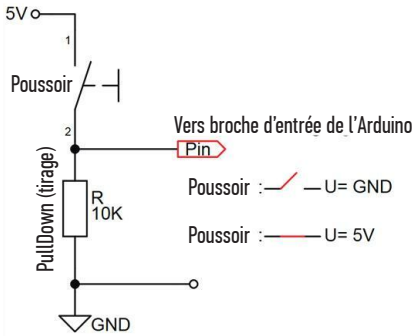


Revenons maintenant à notre configuration, où les broches 8, 9 et 10 sont configurées comme des entrées. Comment vérifier leur comportement ? Pour cela, il faut aller un peu plus loin. Dans la technologie numérique, il n'y a rien de pire qu'un circuit qui est défini comme une entrée et qui n'a rien de connecté. Essayons de comprendre pourquoi en examinant le circuit suivant :

Figure 2-8 ▶
Une entrée ouverte

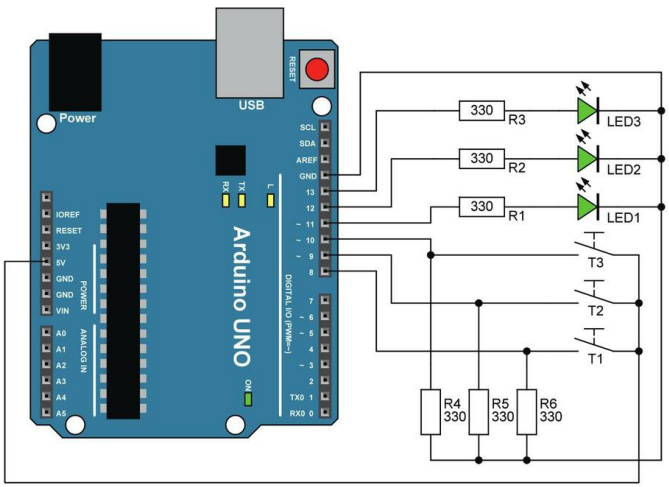


Si l'interrupteur est fermé, la tension de +5 V est présente à la broche d'entrée, et si nous l'ouvrons, vous pensez peut-être être en présence de 0 V. Il n'en est rien, parce qu'une entrée ouverte à laquelle aucun niveau défini n'est affecté reçoit des signaux d'interférences parasites. Il suffit de toucher avec le doigt la broche d'entrée pour injecter un signal de niveau important en constante évolution. Pour cette raison, nous devons utiliser le circuit suivant comportant une résistance de rappel dite *pull-down*.



◀ Figure 2-9
Une entrée ouverte avec résistance pull-down

Si l'interrupteur est fermé, rien ne change par rapport à la situation précédente, le +5 V est appliqué à la broche d'entrée ou tombe sur la résistance pull-down de 10 kΩ reliée à la masse. Mais si l'interrupteur est ouvert, le potentiel de la masse est affecté à la broche d'entrée via la résistance, si bien qu'un niveau LOW de 0 V y est appliqué. Ainsi, dans les deux cas, nous avons un signal défini sur la broche d'entrée. Pour le montage qui suit, nous utiliserons à nouveau la carte Arduino Discoveryboard, qui dispose de résistances pull-down fixes. Le schéma des connexions du prochain montage est le suivant :



◀ Figure 2-10
Le schéma des connexions avec trois LED et trois boutons-poussoirs

Examinons la configuration de test avec les trois boutons-poussoirs correspondants :

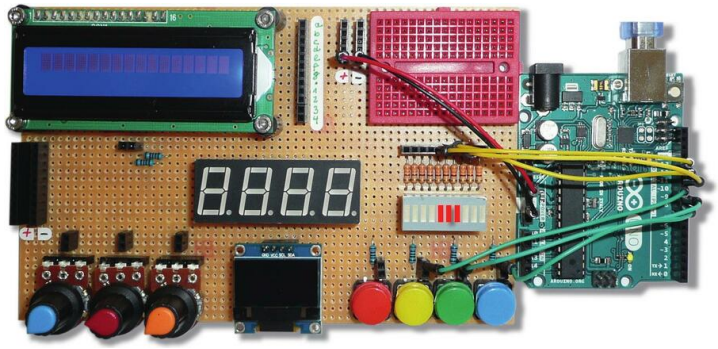


Figure 2-11 ▶
La configuration de test avec des boutons-poussoirs supplémentaires

Nous devons bien sûr modifier la programmation :

```
void setup() {
  DDRB = 0b11111000; // Broches 8, 9, 10 en ENTRÉE. Broches 11, 12,
                     // 13 en SORTIE
  PORTB = 0b00111000; // Broches 11, 12, 13 au niveau HAUT
  Serial.begin(9600); // Interface sérielle à 9600 bauds
}

void loop() {
  // Sortie binaire du registre PINB sur le moniteur série
  Serial.println(PINB,BIN);
  delay(1000); // Pause d'une seconde
}
```

Avant de passer le code en revue, nous devons parler d'un autre registre. Il s'appelle PINx, où *x* représente encore le port.

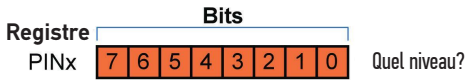


Figure 2-12 ▶
Le registre PINx

Le registre PINB (adresse de broche d'entrée) est essentiellement un registre d'état qui indique l'état du port. Il ne propose qu'un accès en lecture. Nous l'utilisons pour afficher l'état de toutes les broches sur le *moniteur série*. Dans la fonction `setup`, l'interface sérielle est initialisée avec une vitesse de transmission de 9 600 bauds. La sortie est effectuée ultérieurement dans la *loop* (boucle) par la fonction `println` (*print linefeed*, assurant un passage à la ligne suivante) avec le paramètre supplémentaire `BIN` qui produit une sortie binaire de la valeur.

Le moniteur série s'ouvre dans l'environnement de développement via l'icône encadrée en rouge.

Figure 2-13
L'ouverture du moniteur série

Après un clic avec la souris, le moniteur apparaît comme suit :

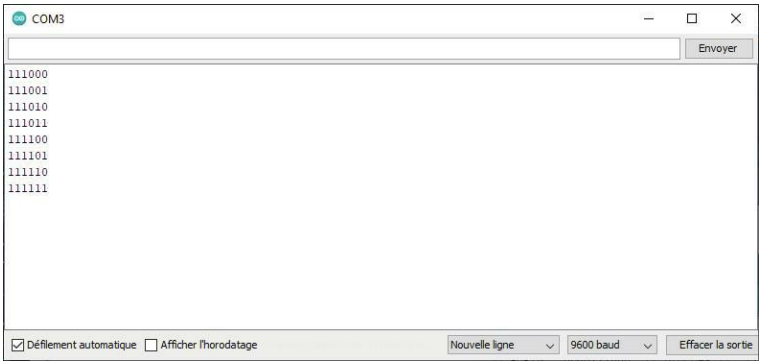


Figure 2-14
La fenêtre du moniteur série

Les trois bits les moins significatifs à droite reflètent l'état des trois interrupteurs. Sur quel(s) bouton(s) ai-je appuyé ici pour obtenir ces combinaisons de bits ?

REMARQUE CONCERNANT LE REGISTRE D'ENTRÉE PINX

Pour ce qui est du registre PINx, toutes les informations d'état du port correspondant sont lues à la fois.

Registres et instructions C++

Pour programmer ou contrôler les différentes broches, ce type de programmation peut s'avérer un peu lourd. Heureusement, on peut utiliser différentes instructions en C ++ qui permettent de nous faciliter la tâche. Nous en parlerons plus en détail dans le montage n° 3, mais je souhaite le mentionner ici dès à présent.

Registre	Instruction C++	Exemple
DDRB	`pinMode`	`pinMode(13, OUTPUT);`
PORTB	`digitalWrite`	`digitalWrite(10, HIGH);`
PINB	`digitalRead`	`digitalRead(8);`

Tableau 2-2
Les registres et leurs équivalents en instructions

Voici quelques informations sur les instructions utilisées.

pinMode

L'instruction `pinMode` programme le sens du flux de données d'une broche numérique. Elle nécessite deux paramètres : le premier spécifie la broche concernée, et le second le sens, où `OUTPUT` correspond à sortie et `INPUT` à entrée.

digitalWrite

L'instruction `digitalWrite` influence l'état d'une broche numérique. Elle nécessite deux paramètres : le premier spécifie la broche concernée, et le second le niveau, où `HIGH` correspond à 5 V et `LOW` à 0 V.

digitalRead

L'instruction `digitalRead` lit l'état d'une broche numérique, dont le résultat est `HIGH` ou `LOW`.

REMARQUE CONCERNANT LE REGISTRE DE SORTIE DDRD

Il existe une méthode plus sûre pour les deux broches RX et TX de l'interface sérielle, situées sur le port D, dont le sens de flux de données ne doit pas être changé. L'exemple suivant programme les broches 2 à 7 comme sorties et ne modifie pas les broches 0 et 1 : `DDRD |= 0b11111100;`

Comment ça marche ? Nous utilisons un opérateur OU binaire avec le registre utilisé. Pour les broches critiques 0 et 1, l'opérateur utilise la valeur 0, ce qui signifie que ces deux bits restent inchangés. Des informations plus détaillées sur la manipulation des bits peuvent être trouvées à l'adresse suivante :

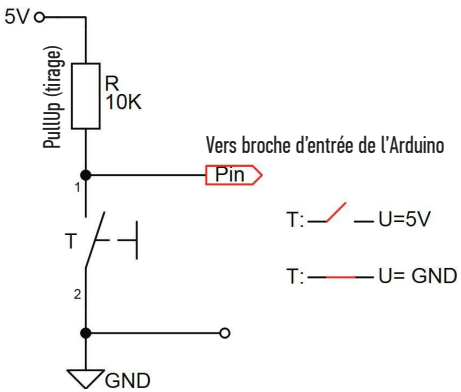
https://playground.arduino.cc/Code/BitMath

Pour obtenir des détails sur le mappage des broches, l'adresse suivante vous sera certainement utile :

https://www.arduino.cc/en/Reference/Atmega168Hardware

La résistance pull-up

Nous avons vu qu'une entrée ouverte sur une broche numérique causait des problèmes et nécessitait par conséquent un circuit externe, utilisant par exemple une résistance pull-down. Mais on peut également prévoir un circuit comportant une résistance pull-up. Il s'agit d'une résistance normale, mais contrairement à la pull-down reliée à la masse, elle est reliée au +5 V de l'alimentation. Dans le cas d'une entrée ouverte, 5 V sont donc appliqués à la broche d'entrée. Considérons le circuit suivant :



◀ Figure 2-15
Une entrée ouverte avec résistance pull-up

Je tiens à faire remarquer que le microcontrôleur contient des résistances pull-up intégrées qui peuvent être activées si nécessaire. Comment ça marche ? Nous devons effectuer deux actions qui sont en réalité contradictoires :

- programmer une broche en entrée ;
- fournir à la broche un niveau HIGH.

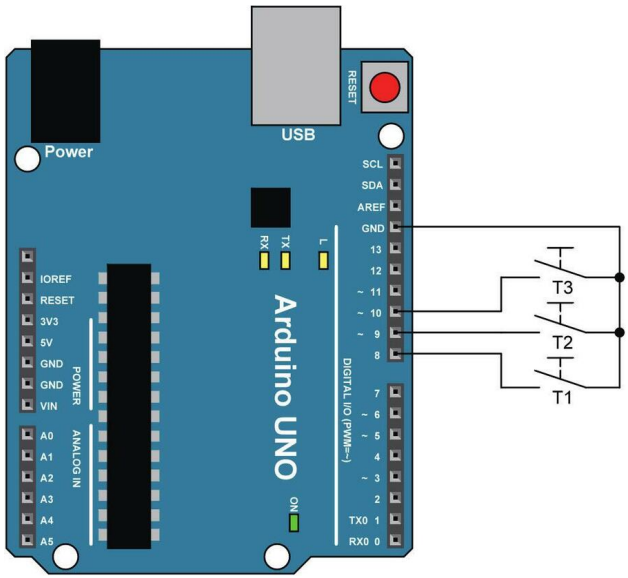
Pourquoi sont-elles contradictoires ? Eh bien, si dans un premier temps une broche est programmée en *entrée*, on s'attend à ce que de l'extérieur un niveau lui soit appliqué puis change si nécessaire. Mais si dans un deuxième temps nous programmons un niveau HIGH pour cette broche, comme si elle était utilisée en *sortie*, la résistance de pull-up interne est alors activée. Regardons le sketch correspondant, qui utilise les registres DDB et PORTB :

```
void setup() {
  DDRB = 0b00000000; // Toutes les broches sont en INPUT
  PORTB = 0b00000111; // Résistances pull-up activées pour les
                      // broches 8, 9 et 10
  Serial.begin(9600); // Initialisation de l'interface sérielle
                      // à 9600 bauds
}

void loop() {
  Serial.println(PINB, BIN);
  delay(500); // Pause de 500 millisecondes
```

Si nous avons maintenant seulement trois interrupteurs miniatures sans résistances, reliés aux entrées numériques 8, 9 et 10, notre circuit fonctionne parfaitement, car les résistances pull-up assurent un niveau correct et sans interférence. Bien sûr, nous ne pouvons pas utiliser pour ce montage les boutons-poussoirs de la carte Arduino Discoveryboard, car ils disposent d'une résistance pull-down fixe. Ce n'est cependant pas un

problème. La carte Arduino Discoveryboard présente une petite plaque de prototypage, sur laquelle vous pouvez très bien enficher les boutons-poussoirs miniatures.

Figure 2-16 ▶
Un circuit avec trois boutons-poussoirs



Si nous ouvrons maintenant le moniteur série, nous verrons que lorsqu'aucun bouton-poussoir n'est pressé, les bits respectifs sont au niveau HIGH, ce qui était à prévoir. Si nous appuyons maintenant sur un bouton-poussoir, le niveau du bit correspondant change de HIGH vers le niveau LOW. Bien entendu, vous pouvez activer ou désactiver les résistances pull-up internes à l'aide d'une commande Arduino classique. Nous en reparlerons.

Problèmes courants

- Vérifiez les fiches sur la plaque de prototypage pour voir si elles correspondent bien au schéma des connexions.

Qu'avez-vous appris ?

- Nous avons expliqué le rapport entre les broches Arduino et les registres internes du microcontrôleur.
- Les registres DDRx, PORTx et PINx permettent de définir et d'interroger la direction du flux de données et les niveaux logiques.

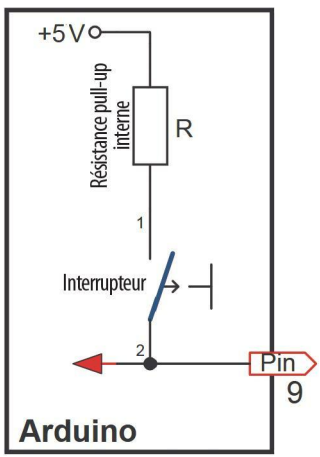
- Nous avons étudié la LED et la manière de la piloter correctement.
- Nous avons évoqué le problème dû à une entrée numérique non connectée et avons décrit une solution utilisant une résistance dite pull-down.
- Des instructions appropriées ont permis d'activer les résistances internes pull-up, de sorte qu'aucun circuit externe comportant des résistances supplémentaires soit nécessaire.

3

Interrogation d'un bouton-poussoir

Manipulation d'une résistance pull-up interne

À quoi peut bien ressembler une résistance pull-up interne ? Examinons le schéma de raccordement suivant :



◀ Figure 3-1
Résistance pull-up interne connectée à la broche numérique 9

J'ai choisi dans cet exemple la broche 9, à laquelle par exemple votre bouton-poussoir est relié. On peut voir que la résistance pull-up R relie la broche 9 à la tension d'alimentation de +5 V via un interrupteur électronique quand celui-ci est fermé. La question est de savoir comment cet interrupteur peut être fermé pour que la broche présente un niveau `HIGH` en l'absence de niveau d'entrée. Les instructions suivantes sont nécessaires, `pin` ayant ici la valeur 9 :

```
pinMode(pin, INPUT);      // Configurer la broche comme entrée
digitalWrite(pin, HIGH); // Activer la résistance pull-up interne
```

Elles ont pour effet de fermer l'interrupteur.

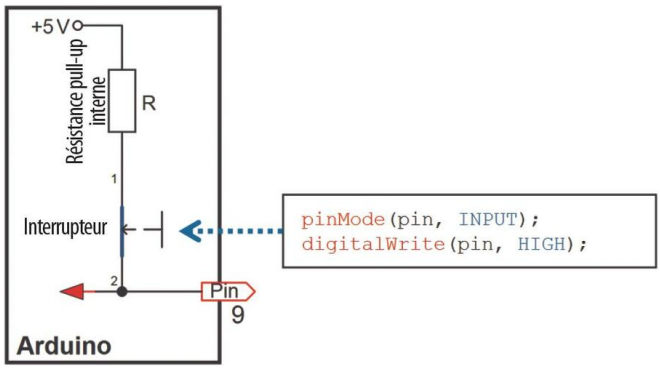


Figure 3-2 ▶ La résistance pull-up interne est activée.

Vous remarquerez que nous configurons une broche numérique en tant qu'entrée parce que nous voulons y raccorder un bouton-poussoir. Jusque-là, ça va. Mais nous essayons ensuite de modifier le niveau de cette broche avec l'instruction `digitalWrite` alors qu'elle n'a pas été configurée en tant que sortie. Qu'est-ce que ça veut dire ?

C'est ça l'astuce. La séquence d'instructions en question permet d'activer la résistance pull-up interne qui possède par ailleurs la valeur de 20K. Cela permet de fixer le potentiel de la broche à +5 V quand l'entrée est ouverte tout en obtenant un niveau d'entrée défini.

RÉSISTANCE PULL-DOWN OU PULL-UP ?

Comme le raccordement d'une broche numérique peut se faire soit à l'aide d'une résistance pull-down externe soit d'une résistance pull-up interne, l'interrogation de la broche ne se programme pas de la même façon. Réfléchissez un peu avant de poursuivre votre lecture. Dans le cas d'une résistance pull-down, le niveau est `LOW` quand l'entrée est ouverte. Pour produire un changement de niveau, il faut appliquer +5 V depuis l'extérieur. Cela signifie que l'interrogation du bouton-poussoir s'effectuera au moyen de la ligne suivante, la variable `buttonState` devant préalablement avoir été initialisée.

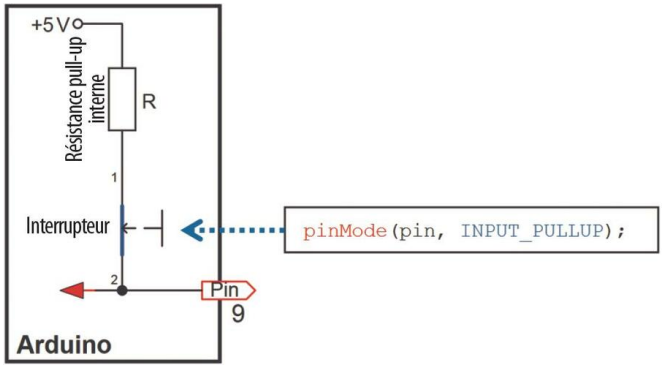
Nous y reviendrons très vite.

```
if(buttonState == HIGH) ...
```

Jusqu'ici, tout va bien. Supposons que vous travaillez maintenant avec une résistance pull-up interne qui génère un niveau HIGH quand le bouton-poussoir est ouvert. Pour produire un changement de niveau, le bouton-poussoir fermé doit appliquer depuis l'extérieur un signal LOW, ou 0 V. L'interrogation du bouton-poussoir s'effectuera alors au moyen de la ligne suivante :

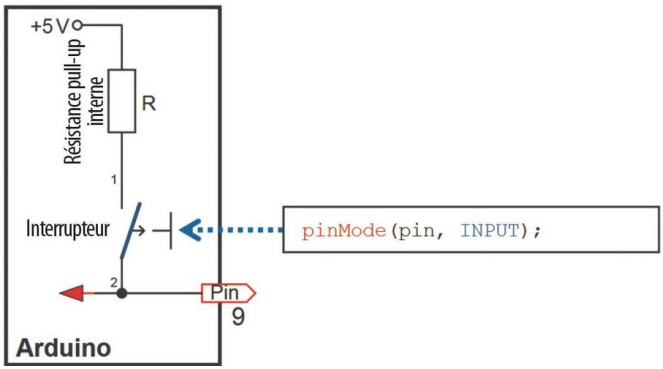
```
if(buttonState == LOW) ...
```

Il existe encore une autre façon de manipuler une résistance pull-up interne. À partir de la version 1.0.1 de l'environnement de développement Arduino, il est possible d'influencer la résistance à l'aide d'un paramètre de mode avec l'instruction `pinMode`.



◀ Figure 3-3
La résistance pull-up interne est activée.

Une seule instruction `pinMode` ferme l'interrupteur. La désactivation s'effectue à l'aide du mode `INPUT` qui rouvre l'interrupteur interne.



◀ Figure 3-4
La résistance pull-up interne est désactivée.

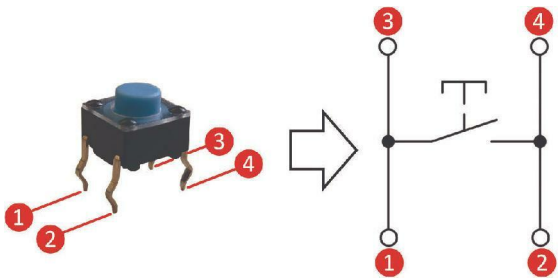
Là aussi, il suffit d'une seule instruction `pinMode` avec le mode indiqué. Nous allons prendre un exemple concret illustrant l'interrogation de l'état

d'une broche numérique. Nous utiliserons l'instruction `digitalRead`. Nous avons déjà vu sa syntaxe dans le montage précédent de la LED clignotante.

Cette fonction n'est pas seulement appelée, elle nous renvoie également une valeur de retour que nous pouvons mettre à profit. La valeur est transmise à la variable `buttonState` au moyen de l'opérateur d'affectation `=`. Les valeurs restituées peuvent être `HIGH` ou `LOW`, qui représentent des constantes prédéfinies par le système – souvenez-vous.

Avant de nous intéresser au sketch, aux composants nécessaires et au schéma de montage correspondant, j'aimerais revenir sur la construction et le fonctionnement d'un bouton-poussoir miniature. La figure suivante présente un bouton-poussoir miniature qui possède quatre pattes. Deux broches sont nécessaires pour fermer un seul contact. Mais ce n'est pas parce que nous disposons de quatre broches que le boîtier contient deux boutons-poussoirs indépendants.

Figure 3-5 ▶
Bouton-poussoir miniature



Sur le schéma de raccordement qui montre le câblage interne, vous pouvez voir que deux pattes sont toujours connectées ensemble. Le bouton est donc accessible au moyen des pattes 1 et 2 ou 3 et 4. Lorsque vous tournez le bouton de 90° et que vous utilisez le même schéma de raccordement, le bouton est constamment fermé. Il faut donc faire particulièrement attention aux pattes qui dépassent sur le côté du boîtier. Elles sont court-circuitées lorsque vous actionnez le bouton. Au besoin, utilisez un multimètre et son testeur de continuité pour identifier avec certitude les contacts du bouton.

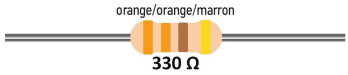
Il existe différents modèles de boutons-poussoirs.

Figure 3-6
Plusieurs types de boutons-poussoirs

Composants nécessaires

Ce montage ne nécessite pas grand-chose et il peut même se passer de composants supplémentaires. En effet, la carte Arduino comporte une LED qui est désignée par la lettre *L*. Toutefois, j'aimerais compléter ce montage par quelques composants que nous réutiliserons pour d'autres montages.

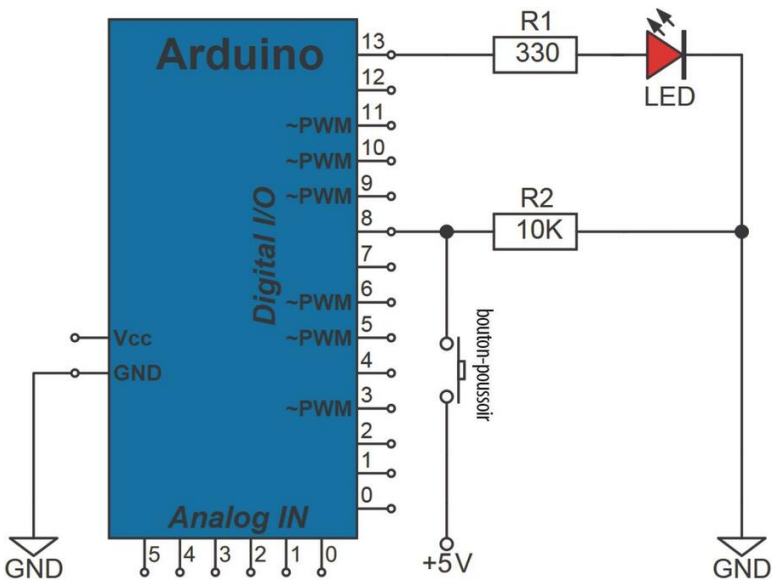
Tableau 3-1
Liste des composants

Composant	
1 LED rouge	
1 résistance de 330 Ω	
1 résistance de 10 kΩ	
1 bouton-poussoir miniature	

Schéma

Le schéma montre une entrée externe assurant un niveau d'entrée sûr grâce à une résistance pull-down externe sur la broche numérique 8.

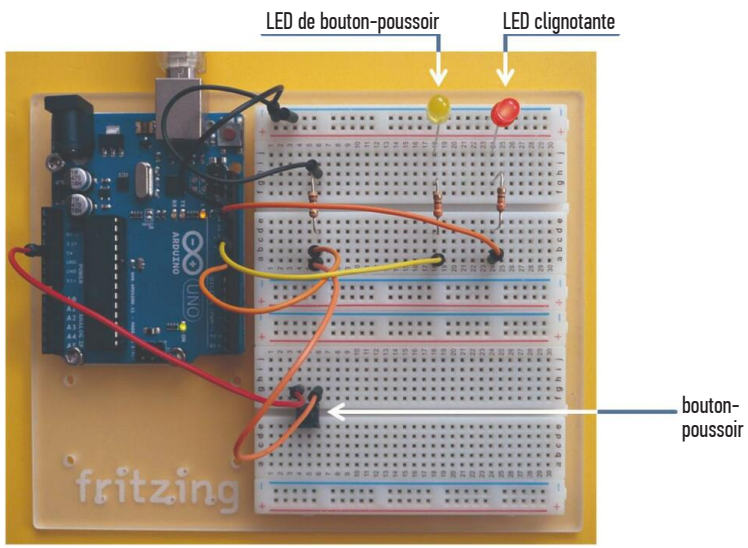
Figure 3-7 ▶
Schéma d'interrogation d'un bouton-poussoir



Réalisation du circuit

Le circuit est facile et rapide à réaliser sur une plaque de prototypage.

Figure 3-8 ▶
Réalisation du circuit d'interrogation d'un bouton-poussoir



Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

Voici le sketch d'interrogation d'un bouton-poussoir et de commande de la LED :

```
int ledPin = 13;          // LED – broche 13
int buttonPin = 8;        // bouton-poussoir – broche 8
int buttonState;          // variable d'état du bouton

void setup() {
  pinMode(ledPin, OUTPUT);   // broche LED comme sortie
  pinMode(buttonPin, INPUT); // broche du bouton-poussoir comme
entrée
}

void loop() {
  buttonState = digitalRead(buttonPin);
  if(buttonState == HIGH)
    digitalWrite(ledPin, HIGH);
  else
    digitalWrite(ledPin, LOW);
}
```

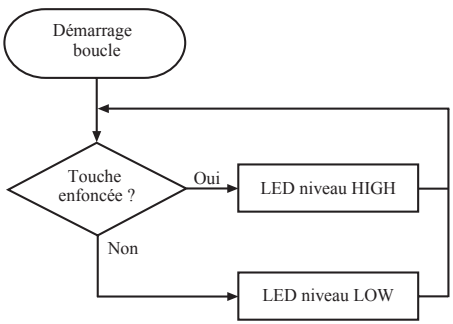
Revue de code

Dans cet exemple, on voit d'emblée qu'on a affaire à plusieurs variables qui doivent être déclarées dès le début. Les explications figurent sous forme de commentaires derrière les lignes d'instructions.

BROCHE NUMÉRIQUE

Une broche numérique opère de manière standard comme entrée et n'a donc pas besoin d'être configurée en tant que telle au moyen de l'instruction **pinMode**. C'est cependant utile pour une meilleure compréhension d'ensemble. Vous pouvez malgré tout laisser tomber cette étape dans la mesure où la quantité de mémoire est limitée et où chaque octet compte.

Examinons le schéma de raccordement suivant :



◀ Figure 3-9
Organigramme pour commander la LED

L'organigramme se lit très facilement. Lorsque l'exécution du sketch arrive à la boucle sans fin `loop`, l'état de la broche du bouton-poussoir est continuellement interrogé et consigné dans la variable `buttonState`. Voici la ligne de code correspondante :

```
buttonState = digitalRead(buttonPin);
```

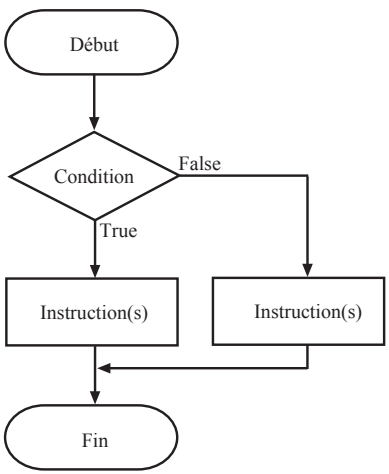
Selon la valeur reçue, l'évaluation est effectuée à la suite de l'interrogation par une structure de contrôle sous la forme d'une décision `if-else` (si-alors-sinon).

```
if(buttonState == HIGH)
    digitalWrite(ledPin, HIGH);
  else
    digitalWrite(ledPin, LOW);
```

L'instruction `if` évalue la condition entre parenthèses, laquelle peut être librement traduite comme suit : « Le contenu de la variable `buttonState` est-il égal à `HIGH` ? Si oui, exécuter la ligne d'instruction qui vient aussitôt après l'instruction `if`. Si non, poursuivre avec l'instruction qui vient après le mot-clé `else`. »

L'organigramme suivant permet de mieux comprendre cette structure de contrôle.

Figure 3-10 ▶ Organigramme pour structure de contrôle `if-else`



Il existe une variante plus simple de la structure de contrôle `if`, dans laquelle la branche `else` est absente. Nous y reviendrons plus tard. Vous voyez donc que le déroulement d'un programme n'est pas forcément linéaire. On peut y insérer des ramifications qui permettent à diverses instructions ou blocs d'instructions d'être exécutés selon différents mécanismes d'évaluation.

Un sketch n'agit pas seulement, mais réagit également à des influences externes, par exemple des signaux de capteur.

OPÉRATEUR D'ÉGALITÉ ET D'AFFECTATION

Une erreur très fréquente chez les débutants consiste à confondre opérateur d'égalité et opérateur d'affectation. L'opérateur d'égalité `==` et l'opérateur d'affectation `=` ont des missions complètement différentes, mais sont souvent utilisés l'un à la place de l'autre. Le pire est que les deux modes d'écriture sont utilisables et valables dans une condition.

Voici l'utilisation correcte de l'opérateur d'égalité :

```
if(buttonState == HIGH)
```

Voici maintenant l'utilisation erronée de l'opérateur d'affectation :

```
if(buttonState = HIGH)
```

Comment se fait-il que ce mode d'écriture ne produise pas d'erreurs ? C'est très simple : il s'ensuit une affectation de la constante `HIGH` (valeur numérique `1`) à la variable `buttonState`. `1` n'étant pas une valeur nulle, elle est interprétée comme étant `true` (vraie). Dans le cas d'une ligne de code `if(true)…`, l'instruction qui suit est toujours exécutée. Une valeur numérique `0` est évaluée comme `false` (fausse) dans C/C++ et toute autre différente de `0` est évaluée comme `true`. De telles erreurs sont difficiles à discerner et cela prend toujours énormément de temps.

Vous trouverez plus d'informations sur les commandes utilisées aux adresses suivantes :

- `if`
  https://www.arduino.cc/en/Reference/If
- `if-else`
  https://www.arduino.cc/en/Reference/Else

Qu'avez-vous appris ?

- Le microcontrôleur de la carte Arduino Uno dispose d'une résistance pull-up interne qui possède la valeur de 20K.
- Ces résistances peuvent être activées ou désactivées soit à l'aide d'une séquence d'instructions `pinMode` ou `digitalWrite`, soit à l'aide d'une seule instruction `pinMode` avec le mode `INPUT_PULLUP` ou `INPUT`.

Clignotement avec gestion des intervalles

Dans le montage n° 1 de commande d'une LED, nous avons vu comment interrompre momentanément l'exécution d'un sketch à un endroit particulier 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 nous faut modifier et compléter le circuit clignotant en y ajoutant quelques composants.

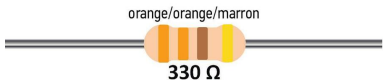
Appuyez sur le bouton-poussoir et il réagit

Que se passerait-il si vous branchiez en plus un bouton-poussoir sur une autre 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

Ce montage nécessite les composants suivants.

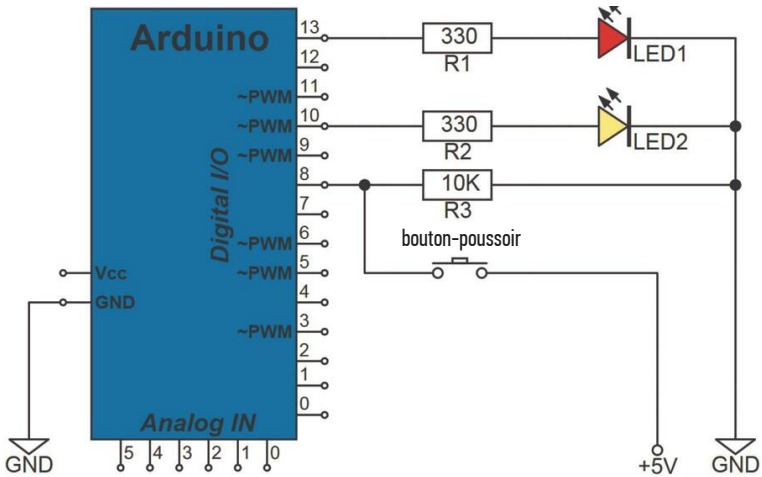
Tableau 4-1 ▶
Liste des composants

Composant	
1 LED rouge	
1 LED jaune	
1 bouton-poussoir miniature	
1 résistance de 330 Ω	
1 résistance de 10K	

Schéma

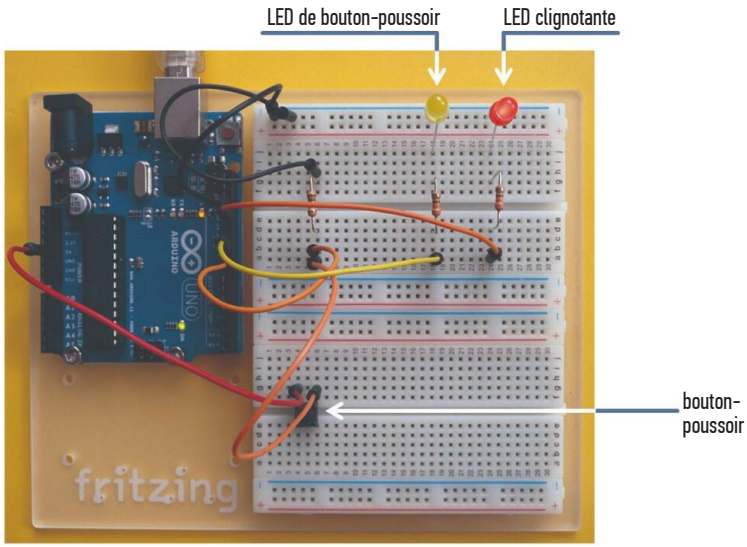
Le schéma montre un niveau d'entrée sûr grâce à une résistance pull-down externe sur la broche numérique 8.

Figure 4-1 ▶
Schéma d'interrogation d'un bouton-poussoir et de commande des LED



Réalisation du circuit

La plaque de prototypage est un peu plus remplie.



◀ Figure 4-2
Réalisation du circuit d'interrogation d'un bouton-poussoir

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

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

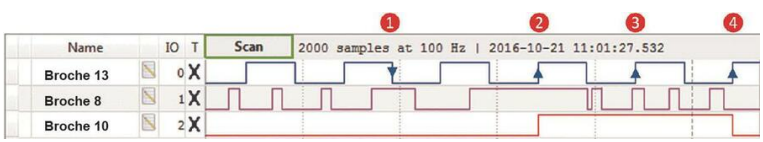
```
// Le code ne fonctionne pas comme prévu
int ledPinBlink  = 13; // LED clignotante rouge – broche 13
int ledPinButton = 10; // LED de bouton-poussoir jaune – broche 10
int buttonPin = 8;     // Bouton-poussoir – broche 8
int buttonState;       // Variable d'état du bouton

void setup() {
  pinMode(ledPinBlink, OUTPUT);  // Broche LED clignotante comme sortie
  pinMode(ledPinButton, OUTPUT); // Broche LED de bouton-poussoir comme
                                 // sortie
  pinMode(buttonPin, INPUT);     // Broche du bouton-poussoir comme entrée
}

void loop() {
  // Faire clignoter la LED clignotante
  digitalWrite(ledPinBlink, HIGH);// LED rouge au niveau HIGH
  delay(1000);                    // Attendre une seconde
  digitalWrite(ledPinBlink, LOW); // LED rouge au niveau LOW
```

```
  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
  else
    digitalWrite(ledPinButton, LOW); // LED jaune au niveau LOW
}
```

Vous remarquerez que l'exécution du sketch 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 programmer un sketch qui réagisse à *tout* moment de son exécution à une action sur le bouton-poussoir et pas seulement à un moment donné quand l'exécution du code atteint l'emplacement correspondant. La fonction `delay` entrave la poursuite du code, donc nous ne pouvons pas l'employer ici. Je vous montre le comportement sur un chronogramme, où figurent les signaux pertinents, ceux de la LED clignotante (broche 13), du bouton-poussoir (broche 8) et de la LED de bouton-poussoir (broche 10).

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



Le signal du haut (ici, en bleu) représente l'état de la LED clignotante sur la broche 13 qui bascule infatigablement chaque seconde entre les niveaux `HIGH` et `LOW`. Le signal suivant (ici, en violet) représente le niveau sur le bouton-poussoir raccordé à la broche 8 qui essaye de commander la LED du bouton-poussoir sur la broche 10. Le dernier signal devrait donc changer de niveau en même temps que le signal sur le bouton-poussoir, mais ce n'est pas le cas. J'ai numéroté quatre points significatifs sur ce chronogramme. Au point 1, je maintiens le bouton-poussoir enfoncé, mais le front descendant du signal ne semble pas être concerné. Pourquoi la LED de bouton-poussoir connectée à la broche 10 ne réagit-elle pas ? C'est très simple ! Il y a deux activations de `delay`. La première fonction `delay` est activée au moment du passage de `HIGH` à `LOW` alors que nous nous trouvons encore dans la deuxième fonction `delay`. L'état du bouton-poussoir n'est donc pas interrogé. Cette interrogation s'effectue uniquement au point 2 où j'ai enfoncé le bouton au moment du changement de niveau de `LOW` à `HIGH`. L'interrogation de la broche numérique 8 à laquelle le bouton-poussoir est connecté est effectuée avant le changement de niveau suivant. L'état

du bouton-poussoir peut maintenant être évalué et la LED connectée à la broche 10 s'allume. Elle le reste jusqu'au moment où le niveau de la LED clignotante passe à `LOW` avec le front montant, ce qui ne se produit qu'au point 4 alors que ce n'était pas le cas au point 3.

C'est la raison pour laquelle nous devons renoncer à utiliser la fonction `delay` et choisir une autre solution que nous montre le sketch suivant. Ne vous en faites pas pour les lignes de code, car nous allons le développer progressivement :

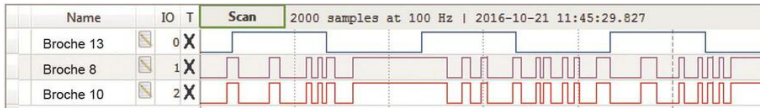
```
// Le code fonctionne comme prévu
int ledPinBlink  = 13; // LED clignotante rouge – broche 13
int ledPinButton = 10; // LED de bouton-poussoir jaune – broche 10
int buttonPin = 8;     // Bouton-poussoir – broche 8
int buttonState;       // Variable d'état du bouton
int interval = 2000;   // Intervalle de temps (2 secondes)
unsigned long prev;    // Variable de temps
int ledStatus = 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 du 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();
    ledStatus = !ledStatus; // Bascule l'état de la LED
    digitalWrite(ledPinBlink, ledStatus); // 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
  else
    digitalWrite(ledPinButton, LOW); // LED jaune au niveau LOW
}
```

Examinons la signification de ce sketch.

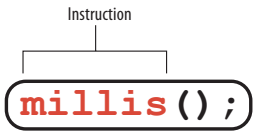
Revue de code

Ce sketch contient quelques variables supplémentaires. Je commencerai par la gestion des intervalles. Le chronogramme représente le comportement du circuit qui est exactement celui que nous voulions :

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



Chaque fois que le bouton-poussoir (broche bouton-poussoir) est enfoncé, le niveau suit la LED du bouton-poussoir et ce, quel que soit l'état ou le changement de niveau de la broche 13 (LED clignotante). Pour la gestion des intervalles, nous avons besoin de la nouvelle fonction `millis` qui se présente comme suit :



La fonction renvoie une valeur en millisecondes depuis le début du sketch. Il faut ici tenir compte d'un aspect 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).

Vous trouverez de plus amples informations sur la fonction `millis` à l'adresse suivante :

https://www.arduino.cc/en/Reference/Millis

QU'EST-CE QU'UN DÉBORDEMENT ?

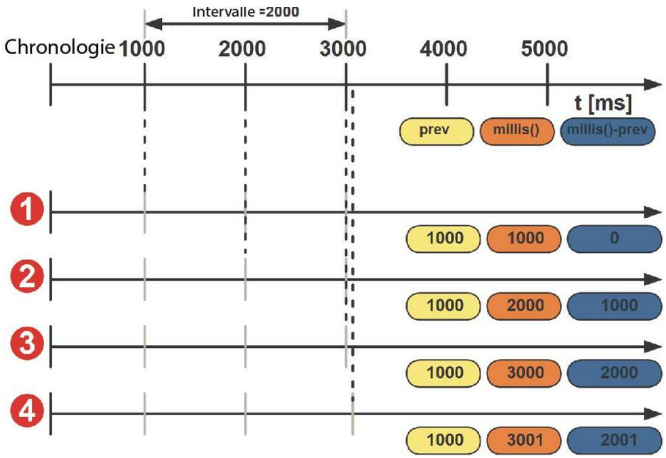
Un débordement signifie pour des variables que le maximum des valeurs que leur type de données peut traiter a été dépassé et va maintenant recommencer à 0. Pour le type de donnée **byte**, qui présente une donnée de 8 bits et peut par conséquent stocker 2^8 = 256 états (de 0 à 255), un débordement se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

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*) ;

- `ledStatus` (la LED clignotante est commandée en fonction de l'état `HIGH` ou `LOW`).

Nous allons maintenant voir comment cela fonctionne. Le diagramme suivant montre une évolution chronologique de la gestion des intervalles.



◀ **Figure 4-5**
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 4-2**
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`. Cela ne se produit qu'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	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. 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 du sketch, 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. Une pression sur le bouton-poussoir est presque simultanément évaluée et affichée.

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 l'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 bouton-poussoir relié au port *8* est finalement interrogé tout à fait normalement et sans retardement.

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

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-poussoirs 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 avant la fin du temps d'attente.
- 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.

Le bouton-poussoir récalcitrant

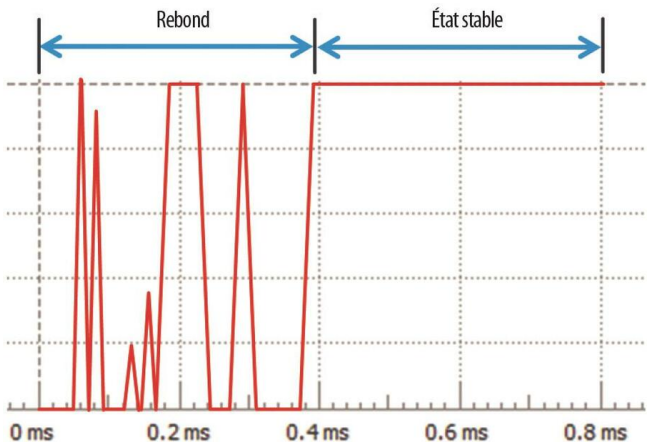
Dans ce montage, vous verrez qu'un bouton-poussoir, ou également un interrupteur, ne se comporte pas toujours comme vous l'auriez voulu. Prenons pour exemple un bouton-poussoir qui, en théorie, ferme le circuit tant qu'il reste enfoncé, et le rouvre quand il est relâché. Rien de neuf en soi et rien de bien difficile à comprendre. Mais les circuits électroniques, dont le rôle consiste par exemple à déterminer le nombre exact de pressions sur le bouton-poussoir pour une exploitation ultérieure, posent un problème dont on ne peut se douter au départ.

Une histoire de rebond

En électromécanique, il existe un effet perturbateur qui se nomme le rebond. Appuyer sur un bouton-poussoir normal et même le maintenir enfoncé revient à fermer le contact mécanique une seule et unique fois dans le bouton-poussoir. Ce n'est pourtant pas le cas la plupart du temps, car le composant en question ouvre et referme plusieurs fois le contact dans un intervalle de temps très court, de l'ordre de la milliseconde. Les surfaces de contact d'un bouton-poussoir ne sont en général pas complètement lisses, et on peut voir de multiples aspérités et impuretés quand on les observe au microscope électronique. Ainsi, les points de contact des matériaux conducteurs ne se touchent pas instantanément et pas durablement au moment du rapprochement. L'effet en question peut aussi être obtenu par vibration ou montage sur ressorts du matériau, le contact étant alors brièvement fermé puis de nouveau ouvert plusieurs fois l'une derrière l'autre lors de la jonction.

Ces impulsions délivrées par le bouton-poussoir sont enregistrées et traitées en bonne et due forme par le microcontrôleur, c'est-à-dire comme si vous appuyiez très vite et très souvent sur le bouton-poussoir. Ce comportement est bien sûr gênant et doit être évité d'une manière ou d'une autre. Regardons maintenant le chronogramme de plus près.

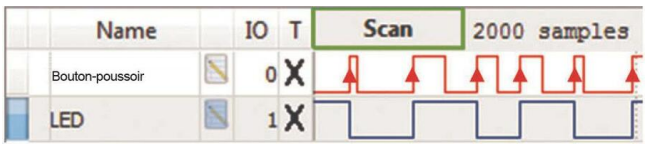
Figure 5-1 ▶
Bouton-poussoir à rebond



J'ai appuyé une seule fois sur le bouton-poussoir et l'ai ensuite maintenu enfoncé, mais celui-ci a interrompu plusieurs fois la liaison souhaitée avant que l'état stable de la connexion ne soit atteint. Cette suite de fermetures et d'ouvertures du circuit jusqu'à ce que le niveau `HIGH` définitif souhaité soit atteint est appelée rebond. Ce comportement peut aussi se constater dans l'opération inverse. Si je relâche le bouton-poussoir, plusieurs impulsions peuvent éventuellement être générées jusqu'à ce que j'obtienne enfin le niveau `LOW` souhaité. Le rebond du bouton-poussoir est à peine perceptible, voire carrément invisible, par l'œil humain et si un circuit censé commander une LED quand le bouton-poussoir est enfoncé était construit, les différentes impulsions se verraient comme un niveau `HIGH` du fait de la persistance oculaire. Essayons donc une autre solution. Nous pourrions construire un circuit avec un bouton-poussoir sur une entrée numérique et une LED sur une autre sortie numérique.

Certes, ce type de rebond ne se voit pas dans un circuit. Mais notre circuit n'est pas le seul composant. Il y a en effet le *matériel,* mais il y a aussi le *logiciel* et nous entendons le configurer de telle sorte que la LED s'allume à la première impulsion. Elle doit s'éteindre à l'impulsion suivante et se

rallumer à celle d'après et ainsi de suite. Nous avons donc affaire à une bascule du niveau logique. Si maintenant plusieurs impulsions sont enregistrées par le circuit ou plutôt par le logiciel quand le bouton-poussoir est pressé, la LED change alors plusieurs fois d'état. Dans le cas d'un bouton-poussoir sans rebond, les états doivent être tels que représentés dans le diagramme de la figure 5-2.

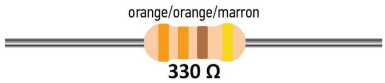


◀ **Figure 5-2**
Changement du niveau de la LED pour un appui sur le bouton-poussoir

On voit que dans le cas de multiples appuis sur le bouton-poussoir (front montant), l'état de la LED bascule.

Composants nécessaires

Ce montage nécessite les composants suivants.

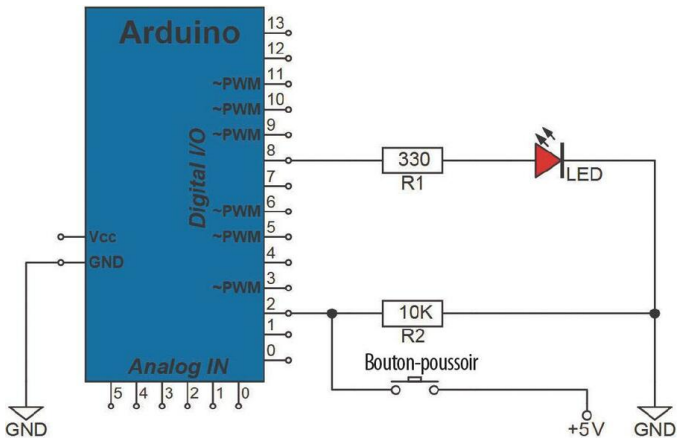
Composant	
1 LED rouge	
1 résistance de 330 Ω	
1 résistance de 10K	
1 bouton-poussoir miniature	

◀ **Tableau 5-1**
Liste des composants

Schéma

Le schéma vous est certainement familier puisqu'il est identique à celui du montage n° 3.

Figure 5-3 ▶
Carte Arduino
avec bouton-poussoir
et LED illustrant le rebond



Réalisation du circuit

Le circuit est identique à celui du montage n° 3. Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

Le sketch suivant change l'état de la LED connectée chaque fois que le bouton-poussoir est enfoncé. Quand le bouton est maintenu enfoncé pendant un temps plus long, l'état ne change plus après la bascule :

```
int buttonPin = 2; // Bouton-poussoir – broche 2
int ledPin = 8;    // LED – broche 8
int ledState;      // État de la LED

int buttonStateActual;  // Enregistre l'état actuel du bouton-poussoir
int prevbuttonState;    // Enregistre l'état précédent du bouton-poussoir
int debouncedButtonState; // Enregistre l'état lissé du bouton-poussoir
int debounceInterval = 50;    // Intervalle de 50 ms
unsigned long lastDebounceTime = 0; // Période sans changement d'état de
                                    // la LED

void setup() {
  pinMode(ledPin, OUTPUT);   // Broche du bouton-poussoir comme sortie
  pinMode(buttonPin, INPUT); // Broche du bouton-poussoir comme entrée
}

void loop() {
  buttonStateActual = digitalRead(buttonPin); // Lit l'état du bouton
```

```
  unsigned long currentTime = millis();      // Lit la période de lissage

  if(buttonStateActual != prevbuttonState) lastDebounceTime = currentTime;
  if(currentTime - lastDebounceTime > debounceInterval){
    if(buttonStateActual != debouncedButtonState){
      debouncedButtonState = buttonStateActual;
     // Changement du niveau de la LED quand l'état du bouton = niveau HIGH
      if(debouncedButtonState == HIGH) ledState = !ledState;
    }
  }
  digitalWrite(ledPin, ledState);      // Commande la LED
  prevbuttonState = buttonStateActual; // Enregistre l'état précédent
                                       // du bouton-poussoir
}
```

Examinons la signification de ce sketch.

Revue de code

Au début de la boucle `loop`, le niveau actuel du bouton-poussoir est toujours consigné dans la variable `buttonStateActual`. La date et l'heure actuelles depuis le début du sketch sont consignées dans la variable `currentTime` avec la fonction `millis`. Si l'état actuel du bouton-poussoir est différent de l'état précédent, la période de lissage est ramenée au nouvel état. Celle-ci est utilisée dans l'interrogation suivante afin de déterminer si l'intervalle prédéfini est écoulé.

```
if(currentTime - lastDebounceTime > debounceInterval){...}
```

Si c'est le cas, l'état courant du bouton-poussoir est comparé à l'état lissé du bouton-poussoir. S'ils sont différents, il faut déterminer si un niveau `HIGH` est présent pour le changement d'état. Ce n'est qu'à cette condition que le changement d'état de la LED aura lieu. Il sera effectué par l'inversion de l'état dans la variable `ledState`.

```
if(debouncedButtonState == HIGH) ledState = !ledState;
```

Enfin, il ne reste plus qu'à commander la LED à l'aide de cette même variable `ledState`

```
digitalWrite(ledPin, ledState);
```

et à transférer l'état actuel du bouton-poussoir dans l'état précédent.

```
prevbuttonState = buttonStateActual;
```

Tout reprend depuis le début.

Problèmes courants

Si la LED ne s'allume pas ou ne bascule pas, plusieurs choses peuvent en être la cause.

- La LED peut avoir été mal polarisée. Rappelez-vous les deux différentes connexions d'une LED que sont l'anode et la cathode.
- La LED est peut-être défectueuse et avoir été grillée par une surtension lors des montages précédents. Testez-la avec une résistance en série sur une source d'alimentation de 5 V.
- Vérifiez les branchements de la LED et des composants sur votre plaque d'essais.
- Vérifiez le sketch que vous avez entré dans l'éditeur de l'IDE. Peut-être avez-vous oublié une ligne ou commis une erreur ou peut-être que le sketch a été mal transmis ?
- Vérifiez le bon fonctionnement du bouton-poussoir utilisé avec un testeur de continuité ou un multimètre.

Qu'avez-vous appris ?

- Vous savez maintenant que des composants mécaniques tels que des boutons-poussoirs ou des interrupteurs ne se ferment ou ne s'ouvrent pas immédiatement. Plusieurs brèves interruptions successives peuvent résulter par exemple de tolérances de fabrication, d'impuretés ou de matériel en vibration, avant d'arriver à un état stable. Ce comportement est enregistré et traité comme tel par des circuits électroniques. Si, par exemple, vous devez compter le nombre d'appuis sur le bouton-poussoir, ces impulsions multiples peuvent s'avérer extrêmement gênantes.
- Ce comportement peut être corrigé de différentes manières :
  - par une solution logicielle (par exemple, une stratégie de temporisation lors de l'interrogation du signal d'entrée) ;
  - par une solution matérielle (par exemple, un circuit RC).

Le séquenceur de lumière

Vous maîtrisez déjà suffisamment les LED pour être en mesure de réaliser des montages où clignotent plusieurs diodes électroluminescentes. Ça n'a l'air de rien dit comme ça, mais ce n'est pas si simple. Nous allons commencer par un séquenceur de lumière, qui commande une par une différentes LED. Souvenez-vous du premier montage pour lequel nous avions programmé un effet similaire. Toutefois, l'objectif était simplement de commander une LED en utilisant les ports ou la manipulation des registres correspondants. Dans ce montage, nous parviendrons à un résultat similaire, mais par un autre biais.

C'est chacun son tour

Les LED branchées sur les broches numériques doivent s'allumer conformément au modèle présenté ci-dessous.

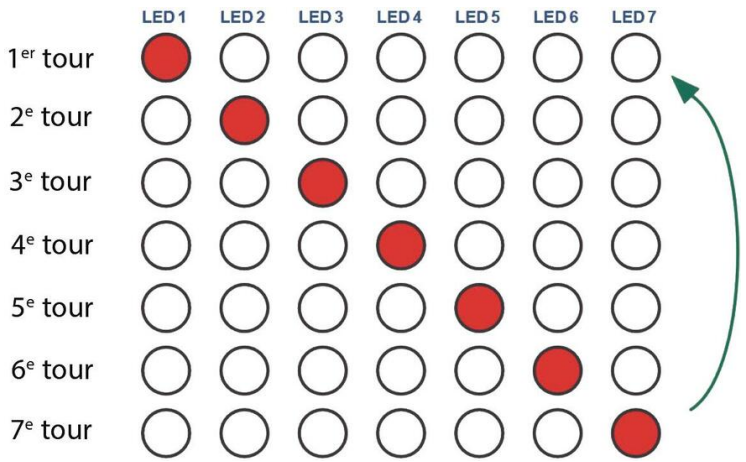


Figure 6-1
Séquence d'allumage des 7 LED

À chaque tour, la LED s'allume une position plus loin à droite. Arrivé à la fin, le cycle reprend au début. Vous pouvez programmer les diverses broches, qui toutes sont censées servir de sortie, de différentes manières. Dans l'état actuel de vos connaissances, vous devez déclarer sept variables et les initialiser avec les valeurs de broche correspondantes. Ce qui pourrait donner ceci :

```
int ledPin1 = 7;
int ledPin2 = 8;
int ledPin3 = 9;
…
```

Chaque broche doit être ensuite programmée dans la fonction `setup` avec `pinMode` comme sortie, ce qui représente aussi un travail de saisie considérable :

```
pinMode(ledPin1, OUTPUT);
pinMode(ledPin2, OUTPUT);
pinMode(ledPin3, OUTPUT);
…
```

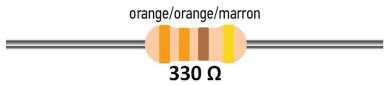
Voici donc la solution. Je voudrais vous présenter un type intéressant de variable, capable de mémoriser plusieurs valeurs du même type de donnée sous un même nom.

Mais comment une variable peut-elle mémoriser plusieurs valeurs sous un seul et même nom ? Et comment faire pour sauvegarder ou appeler les différentes valeurs ?
Patience ! C'est possible. Cette forme spéciale de variable est appelée tableau *(array)*. On n'y accède pas seulement par son nom évocateur, car une telle variable possède aussi un index. Cet index est un nombre entier incrémenté. Ainsi, les différents éléments du tableau – c'est le nom donné aux valeurs stockées – peuvent être lus ou modifiés. Vous allez voir comment dans le code du sketch ci-après.

Composants nécessaires

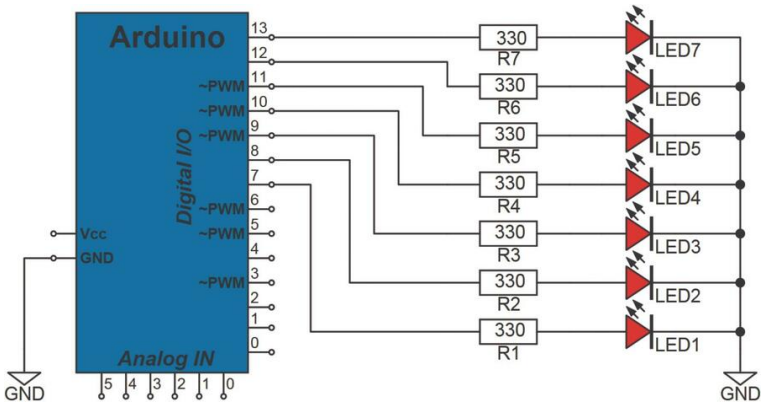
Ce montage nécessite les composants suivants.

Tableau 6-1 ▶
Liste des composants

Composant	
7 LED rouges	
7 résistances de 330 Ω	

Schéma

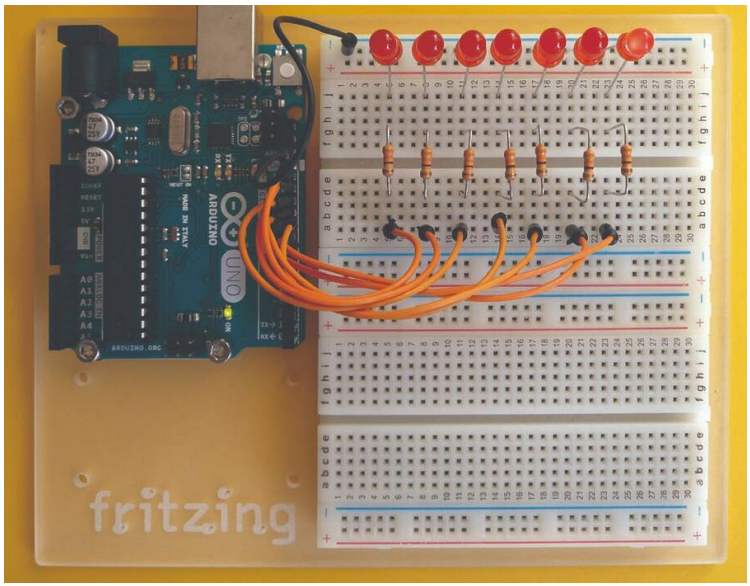
Le schéma vous est certainement familier puisqu'il est identique à celui du montage n° 3.



◀ Figure 6-2
Carte Arduino commandant 7 LED pour un séquenceur de lumière

Réalisation du circuit

Le circuit ressemble beaucoup à celui du premier montage.



◀ Figure 6-3
Réalisation du circuit de séquenceur de lumière à 7 LED

ATTENTION AUX BRANCHEMENTS !

Quand vous branchez des composants électroniques tout près les uns des autres, comme c'est ici le cas, soyez très attentif, car il arrive souvent de se tromper et d'occuper le trou voisin sur la plaque, si bien que le circuit ne fonctionne qu'en partie, voire pas du tout. Cela devient sérieux si vous travaillez avec les lignes d'alimentation et de masse placées l'une à côté de l'autre. Des problèmes peuvent aussi résulter de cavaliers flexibles mal enfoncés dans leur trou, dont les fils conducteurs dénudés ressortent en partie. Des courts-circuits peuvent se produire quand on bouge ces cavaliers, lesquels peuvent tout abîmer. Il faut donc se montrer soigneux.

Sketch Arduino

Voici le code du sketch pour commander le séquenceur de lumière à 7 LED :

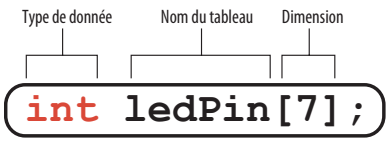
```
int ledPin[] = {7, 8, 9, 10, 11, 12, 13}; // Tableau de LED avec
                                          // numéros de broche
int waitTime = 200; // Pause entre les changements en ms

void setup() {
  for(int i = 0; i < 7; i++)
    pinMode(ledPin[i], OUTPUT); // Toutes les broches du tableau
                                // comme sorties
}

void loop() {
  for(int i = 0; i < 7; i++) {
    digitalWrite(ledPin[i], HIGH); // Élément de tableau au niveau
                                   // HIGH
    delay(waitTime);               // Une pause entre les changements
    digitalWrite(ledPin[i], LOW);  // Élément de tableau au niveau
                                   // LOW
  }
}
```

Examinons la signification de ce sketch, car ce programme comporte quelques nouveautés.

Revue de code

Dans le sketch du séquenceur de lumière, vous rencontrez pour la première fois un tableau et une boucle. Cette dernière est nécessaire pour accéder facilement aux différents éléments du tableau par le biais des numéros de broche. D'une part les broches sont toutes programmées en tant que sorties, et d'autre part les sorties numériques sont sélectionnées. L'accès à

chaque élément se fait par un index et comme la boucle utilisée ici dessert automatiquement un certain domaine de valeurs, cette construction est idéale pour nous. Commençons par la variable de type `array` (tableau). La déclaration ressemble à celle d'une variable normale, à ceci près que le nom doit être suivi d'une paire de crochets.

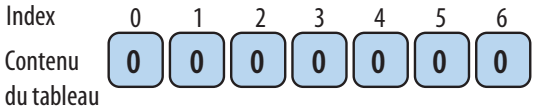


◀ **Figure 6-4**
Déclaration du tableau

Déclaration du tableau

- Le type de donnée définit quel type les différents éléments du tableau doivent avoir.
- Le nom du tableau est un nom évocateur pour accéder à la variable.
- Le nombre entre les crochets indique combien d'éléments le tableau doit contenir.

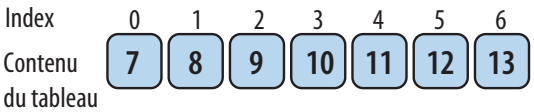
Vous pouvez imaginer un tableau comme un meuble à plusieurs tiroirs. Chaque tiroir est surmonté d'une étiquette portant un numéro d'ordre. Si je vous donne par exemple pour instruction d'ouvrir le tiroir numéro 3 et de regarder ce qu'il y a dedans, les choses sont plutôt claires non ? Il en va de même pour le tableau.



Tous les éléments de ce tableau ont été implicitement initialisés avec la valeur `0` après la déclaration. L'initialisation peut toutefois être faite de deux manières différentes. Nous avons choisi la manière facile et les valeurs, dont le tableau est censé être pourvu, sont énumérées derrière la déclaration entre deux accolades et séparées par des virgules :

```
int ledPin[] = {7, 8, 9, 10, 11, 12, 13};
```

Sur la base de cette ligne d'instruction, le contenu du tableau est le suivant.



N'avons-nous pas oublié quelque chose d'important ? Dans la déclaration du tableau, il n'y a rien entre les crochets. La taille du tableau devrait pourtant y être indiquée.

C'est vrai, mais le compilateur connaît déjà dans le cas présent – par les informations fournies pour l'initialisation faite dans la même ligne – le nombre d'éléments. Aussi la dimension du tableau n'a-t-elle pas besoin d'être indiquée. L'initialisation, quelque peu fastidieuse, consiste à affecter explicitement les différentes valeurs à chaque élément du tableau :

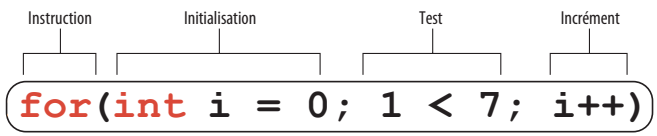
```
int ledPin[7]; // Déclaration du tableau avec 7 éléments
void setup() {
  ledPin[0] = 7;
  ledPin[1] = 8;
  ledPin[2] = 9;
  ledPin[3] = 10;
  ledPin[4] = 11;
  ledPin[5] = 12;
  ledPin[6] = 13;
  // ...
}
```

INDEX D'UN TABLEAU

L'index du premier élément du tableau est toujours le chiffre **0**. Si, par exemple, vous déclarez un tableau de 10 éléments, l'index admis le plus élevé sera le chiffre **9** – soit toujours un de moins que le nombre d'éléments. Si vous ne vous en tenez pas à cette règle, vous pouvez provoquer une erreur à l'exécution que le compilateur de l'environnement de développement ne détecte ni au moment du développement ni plus tard pendant l'exécution. C'est pourquoi vous devez redoubler d'attention !

Venons-en maintenant à la boucle et regardons la syntaxe de plus près.

Figure 6-5 ▶
Boucle `for`



La boucle introduite par le mot-clé `for` est appelée boucle `for`. Suivent entre parenthèses certaines informations sur les caractéristiques fondamentales.

- À partir de quelle valeur la boucle doit-elle commencer à compter ? *(initialisation)*
- Jusqu'à combien doit-elle compter ? *(test)*
- De combien la valeur initiale doit-elle être modifiée ? *(incrément)*

Ces trois informations déterminent le fonctionnement de la boucle `for` et définissent son comportement au moment de l'appel.

QUAND UTILISER UNE BOUCLE FOR ?

Une boucle `for` est utilisée la plupart du temps quand on connaît au départ le nombre de fois que certaines instructions doivent être exécutées. Ces caractéristiques-clés sont définies dans ce qui est appelé l'en-tête de boucle, qui correspond à ce qui est inclus entre parenthèses.

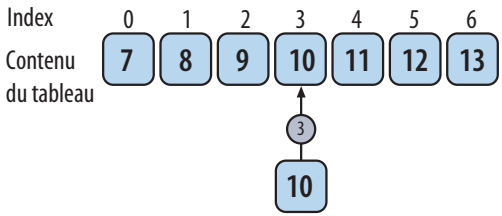
Mais soyons plus concrets. La ligne de code suivante :

```
for(int i = 0; i < 7; i++)
```

déclare et initialise une variable `i` du type `int` avec la valeur `0`. L'indication du type de donnée dans la boucle stipule qu'il s'agit d'une variable locale qui n'existe que tant que la boucle `for` itère, c'est-à-dire suit son cours. La variable `i` est effacée de la mémoire à la sortie de la boucle. Le nom exact d'une telle variable dans une boucle est « variable de contrôle ». Elle parcourt une certaine zone tant que la condition (`i < 7`) – désignée ici sous le nom de « test » – est remplie. Une mise à jour de la variable est ensuite effectuée selon l'expression de l'incrément. L'expression `i++` ajoute la valeur `1` à la variable `i`.

Les signes `++` sont un opérateur qui ajoute la valeur `1` au contenu de l'opérande, donc à la variable. Les programmeurs sont paresseux de naissance et font tout pour formuler au plus court ce qui doit être tapé. Quand on pense au nombre de lignes de code qu'un programmeur doit taper dans sa vie, moins il y a de caractères et mieux c'est. Il s'agit aussi à terme de consacrer plus de temps à des choses plus importantes – par exemple encore plus de code – en adoptant un mode d'écriture plus court. Toujours est-il que les deux expressions suivantes ont exactement le même effet : `i++;` et `i = i + 1;`

Deux caractères de moins ont été utilisés, ce qui représente tout de même une économie de 40 %. Mais revenons-en au texte. La variable de contrôle `i` sert ensuite de variable d'index dans le tableau et traite ainsi l'un après l'autre les différents éléments de ce tableau.



Sur cette capture d'écran d'une itération de la boucle, la variable `i` présente la valeur `3` et a donc accès au 4e élément dont le contenu est `10`. Autrement dit, toutes les broches consignées dans le tableau `ledPin` sont programmées en tant que sorties dans la fonction `setup` au moyen des deux lignes suivantes :

```
for(int i = 0; i < 7; i++)
  pinMode(ledPin[i], OUTPUT);
```

Une chose importante encore : si, dans une boucle `for`, il n'y a aucun bloc d'instructions, formé au moyen d'accolades (comme nous en verrons un bientôt dans la fonction `loop`), seule la ligne venant immédiatement après la boucle `for` est prise en compte par cette dernière. Le code de la fonction `loop` contient seulement une boucle `for` dont la structure de bloc donne cependant accès à plusieurs instructions :

```
for(int i = 0; i < 7; i++) {
  digitalWrite(ledPin[i], HIGH); // Élément de tableau au niveau HIGH
  delay(waitTime);
  digitalWrite(ledPin[i], LOW);  // Élément de tableau au niveau LOW
}
```

Je voudrais vous montrer dans un court sketch comment la variable de contrôle `i` est augmentée (incrémentée) :

```
void setup() {
  Serial.begin(9600);  // Configuration de l'interface série
  for(int i = 0; i < 7; i++)
    Serial.println(i); // Affichage sur l'interface série
}

void loop(){ /* vide */ }
```

Puisque notre Arduino n'a pas de fenêtre d'affichage, nous devons trouver autre chose. L'interface série sur laquelle il est fréquemment branché peut nous servir à envoyer des données. L'environnement de développement dispose d'un moniteur série capable de recevoir et d'afficher ces données sans problème. Vous pouvez même l'utiliser pour envoyer des données à

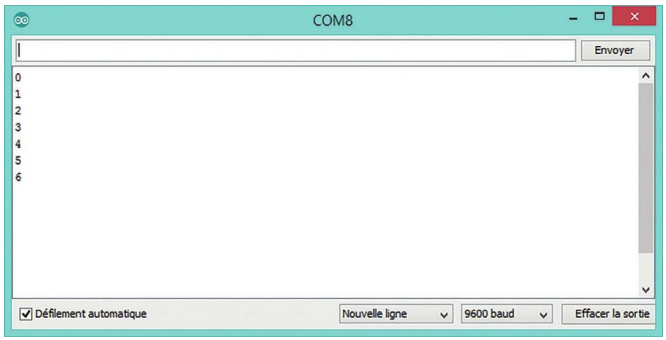
la carte Arduino. Vous en saurez plus bientôt. Le code initialise par l'instruction suivante :

```
Serial.begin(9600);
```

l'interface série avec une vitesse de transmission de 9 600 bauds. La ligne suivante :

```
Serial.println(i);
```

envoie ensuite au moyen de la fonction `println` la valeur de la variable `i` à l'interface. Il ne vous reste plus qu'à ouvrir le moniteur série pour afficher les valeurs de la figure 6-6 :



◀ Figure 6-6
Affichage des valeurs dans le moniteur série

On voit ici comment les valeurs de la variable de contrôle `i`, dont nous avons besoin dans notre sketch pour sélectionner les éléments du tableau, sont affichées de `0` à `6`. J'ai placé le code dans la fonction `setup` pour que la boucle `for` ne soit exécutée qu'une fois et ne s'affiche pas constamment. La figure 6-7 page suivante montre de plus près les différents passages de la boucle `for`.

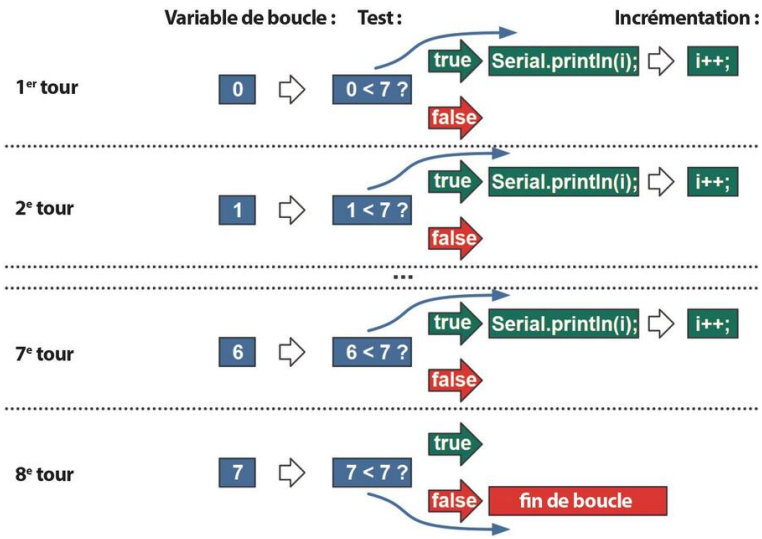


Figure 6-7 ▶ Comportement de la boucle `for`

Il me faut maintenant parler de la programmation orientée objet, car elle va me servir à vous expliquer la syntaxe. Nous reviendrons plus tard sur ce mode de programmation puisque C++ est un langage orienté objet (ou OOP sous sa forme abrégée). Ce langage est tourné vers la réalité constituée d'objets réels tels que par exemple table, lampe, ordinateur, barre de céréales, etc. Aussi les programmeurs ont-ils défini un « objet » représentant l'interface série. Ils ont donné à cet objet le nom de `Serial`, et il est utilisé à l'intérieur d'un sketch. Chaque objet possède cependant d'une part certaines caractéristiques (telles que la couleur ou la taille) et d'autre part un ou plusieurs comportements qui définissent ce qu'on peut faire avec cet objet. Dans le cas d'une lampe, le comportement serait par exemple le fait de s'allumer ou de s'éteindre. Mais revenons à notre objet `Serial`. Le comportement de cet objet est géré par de nombreuses fonctions qui sont appelées méthodes en programmation orientée objet (OOP). Deux de ces méthodes vous sont déjà familières : la méthode `begin` qui initialise l'objet `Serial` avec le taux de transmission voulu, et la méthode `println` (`print line` signifie en quelque sorte imprimer/afficher avec un saut de ligne) qui envoie quelque chose sur l'interface série. Le lien entre objet et méthode est assuré par l'opérateur point (`.`) qui les relie ensemble. Quand je dis par conséquent que `setup` et `loop` sont des fonctions, ce n'est qu'une demi-vérité car il s'agit, à bien y regarder, de méthodes.

Vous trouverez plus d'informations sur la programmation de l'interface série aux adresses suivantes :

https://www.arduino.cc/en/Reference/Serial

https://www.arduino.cc/en/Serial/Begin

https://www.arduino.cc/en/Serial/Println

L'INTERFACE SÉRIE POUR LA RECHERCHE D'ERREURS

Vous savez maintenant comment envoyer quelque chose à l'interface série. Vous pouvez vous en servir pour trouver une ou plusieurs erreurs dans un sketch. Si le sketch ne fonctionne pas comme prévu, placez des instructions d'écriture sous forme de `Serial.println()` à divers endroits qui vous paraissent importants dans le code et affichez certains contenus de variable ou encore des textes. Vous pouvez ainsi savoir ce qui se passe dans votre sketch et pourquoi il ne marche pas bien. Vous devez seulement apprendre à interpréter les données affichées. Ce n'est pas toujours facile et il faut un peu d'entraînement.

Manipulation des registres

Dans le montage n° 2 sur la programmation de bas niveau de la carte Arduino, nous avons vu qu'il est très facile d'influencer les broches numériques par la manipulation des registres. Cela peut se révéler utile pour notre séquenceur de lumière. Le circuit suivant ne possède que 6 LED avec les résistances série correspondantes, chaque LED étant associée à un numéro écrit en rouge. Nous allons bientôt voir à quoi ça sert.

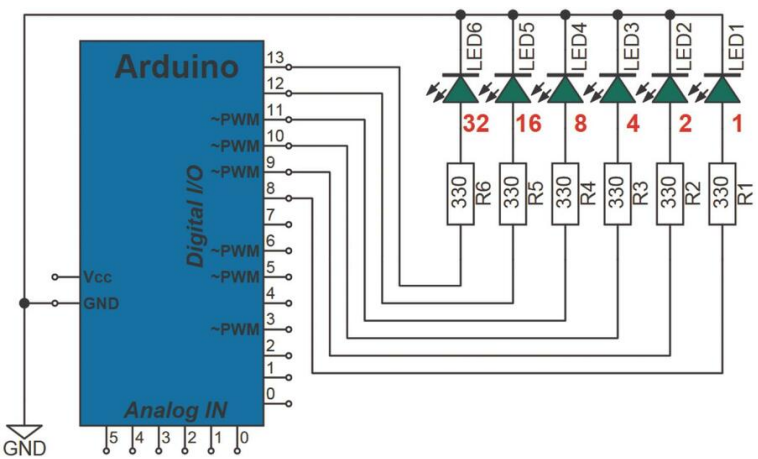
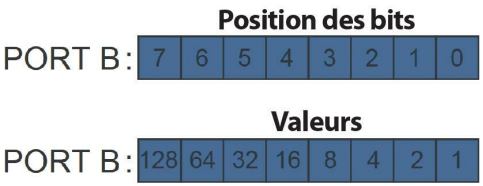


Figure 6-8
Circuit du petit séquenceur de lumière

Examinons le registre du PORT B :



Les huit bits d'un port sont regroupés dans un octet. Chacun des bits de cet octet possède un numéro croissant de droite à gauche en commençant à 0, comme vous pouvez le voir dans la rangée du haut. Cela nous permet d'adresser chaque bit de façon univoque de 0 à 7. En plus de sa position à l'intérieur de l'octet, chaque bit possède une valeur qui dépend aussi de cette position et qui augmente également de droite à gauche. La valeur de chaque bit est indiquée dans la rangée du bas. Reste à savoir comment nous en sommes arrivés à ces valeurs. C'est simple ! Le système binaire connaissant uniquement les états 0 et 1, la base de calcul de la valeur est le chiffre 2. Comme notre système décimal utilise les chiffres 0 à 9, il y a donc 10 états possibles. Par conséquent, la base de calcul de la valeur est le chiffre 10. Mais revenons-en à notre système binaire. Comment calcule-t-on la valeur ? On utilise l'équation suivante :

$$\text{Valeur} = 2^{\text{Position}}$$

Exemple : quelle est la valeur du bit à la position 4 ? L'équation est la suivante :

$$\text{Valeur} = 2^4 = 16$$

Examinons maintenant la variante de syntaxe obtenue par manipulation de registre ou de bit. Pour créer le motif de LED voulu, il faut simplement écrire un 1 aux emplacements de l'octet où les LED doivent être allumées :

Le code du sketch est alors le suivant :

```
void setup() {
  DDRB  = 0b11111111; // PORT B entièrement défini comme SORTIE
  PORTB = 0b00010101; // Création du motif de LED
}

void loop() { /* vide */ }
```

Nous voyons que la ligne contenant la combinaison de bits suivante :

```
PORTB = 0b00010101;
```

correspond précisément au motif de LED. Il ne faut évidemment pas utiliser de nombre binaire lors de l'initialisation. Cela fonctionne aussi avec un nombre entier. Faisons une petite expérience en affichant des combinaisons de bits comprises entre 0 et 63. Il suffit d'incrémenter – c'est-à-dire d'augmenter – constamment une valeur de 1. Mais cette fois, une partie

du code doit se trouver à l'intérieur de la fonction `loop`, car la modification doit avoir lieu à intervalles réguliers.

```
byte pattern = 0;
void setup() {
  DDRB  = 0b11111111; // PORT B entièrement défini comme SORTIE
}

void loop() {
  PORTB = pattern++; // Incrémenter le motif
  delay(100);       // Courte pause de 100 ms
}
```

La variable `pattern` est de type `byte`. Elle compte donc huit bits et peut stocker des valeurs comprises entre 0 et 255. Quand nous l'incrémentons à l'intérieur de la fonction `loop`, un débordement se produira tôt ou tard, quand le contenu de la variable atteindra 255 et que la valeur 1 y sera ajoutée encore une fois. Ça ne fonctionnera pas, car la capacité de stockage sera épuisée. Il se produit donc un débordement et le cycle reprendra au début. La ligne suivante :

```
PORTB = pattern++;
```

incrémente la variable. L'*opérateur d'incrémentation* qui est représenté par les deux signes + est chargé d'augmenter la valeur. Le même résultat est obtenu avec les lignes suivantes :

```
pattern = pattern + 1;
PORTB = pattern;
```

C'est évidemment une question de goût, mais les programmeurs préfèrent les formulations aussi courtes que possible afin d'avoir à saisir moins de code. Nous en arrivons à une partie très intéressante : la *manipulation de bits*. Les bits peuvent être manipulés de diverses façons à l'aide d'opérateurs, même si cela demande un peu d'expérience. Une fois que l'on maîtrise ce type de programmation, on peut s'amuser à combiner les bits à sa guise. Ce montage consiste à commander un anneau de LED qui nous resservira plus tard. Nous allons voir comment faire circulerl'allumage d'une LED. Pour cela, nous utiliserons les *opérateurs de bits*.

Opérateurs de décalage

Pour décaler une LED d'un bit au suivant, nous utilisons l'un des opérateurs de décalage.

- >> : décaler vers la droite
- << : décaler vers la gauche

Comment ça marche ? Examinons le contenu de PORT B et son évolution au fil du temps :

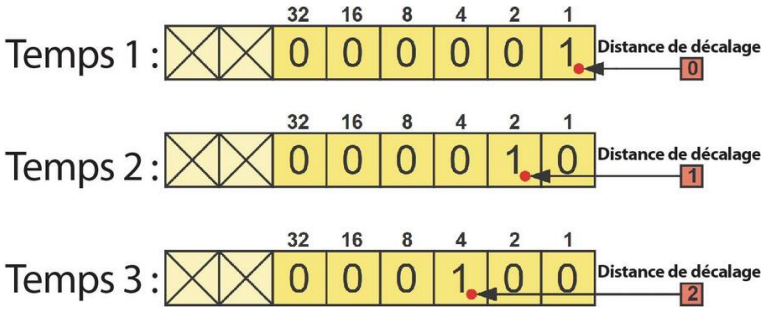


Figure 6-9 ▶
Contenu du registre de PORT B à différents moments

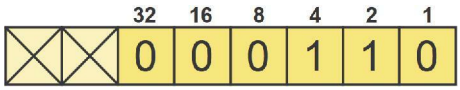
On constate que le chiffre `1` qui se trouve initialement à l'extrême droite se décale peu à peu vers la gauche. Le petit point rouge indique la position finale à un moment donné. Mais comment décale-t-on ce `1` de droite à gauche ? L'opérateur de décalage vers la gauche va nous être très utile. Nous présupposons que le 1 qui se trouve à l'extrême droite et qui, soit dit en passant, se nomme *LSB (Least Significant Bit)* c'est-à-dire bit de poids faible, sert toujours de position de départ pour toutes les opérations de décalage. Le code de sketch suivant exécute la fonction de décalage :

```
byte pos = 0; // Valeur de position
void setup() {
  DDRB  = 0b11111111; // PORT B entièrement défini comme SORTIE
}

void loop() {
  PORTB = 1 << pos++; // Décaler le "1" vers la gauche
  if(pos > 5) pos = 0;
  delay(500);       // Courte pause de 500 ms
}
```

La variable apparaît dans la fonction en tant qu'indicateur de distance de décalage. Au départ, elle intègre la valeur 0, ce qui signifie qu'à la première itération de la boucle, le 1 ne change pas de position, comme c'est le cas au marqueur temporel Temps 1. Après le traitement de l'instruction, la variable `pos` est augmentée de la valeur `1`, ce qui signifie qu'à la prochaine itération de la boucle, il se produira un décalage d'une position vers la gauche. Notez que le décalage vers la gauche fait apparaître un 0 sur côté droit. Il en va de même à chaque nouvelle itération. Lorsque la valeur `pos` est supérieure à 5, ce qui dépasse les possibilités d'affichage de notre dispositif à LED à six bits, elle est ramenée à la valeur `0`. Le cycle reprend alors au début.

Placer des bits

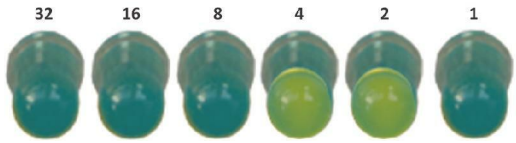
Nous allons maintenant voir comment placer des bits, c'est-à-dire leur affecter la valeur 1 sans influencer l'état actuel des bits existants. Examinons la situation suivante dans laquelle la combinaison de bits présentée représente le masque de bits (suite de bits) de départ :



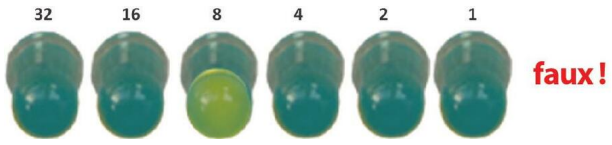
Cette fois, nous voulons décaler le bit ayant la valeur 8 jusqu'à la position 3. Pour y parvenir, nous pouvons écrire le sketch suivant :

```
void setup() {
  DDRB  = 0b11111111; // PORT B entièrement défini comme SORTIE
  PORTB = 0b00000110; // Masque de bits de départ
  delay(500);
  PORTB = 1 << 3;     // Décaler de 3 positions vers la gauche
}

void loop() { /* vide */ }
```

Le masque de bits de départ qui reste visible pendant 500 ms est le suivant :



Ensuite, la LED ayant la valeur 8 doit s'allumer en plus de celles qui sont déjà éclairées. Quel résultat obtient-on ? Voyez par vous-même :



Quel est le problème ? Nous avons placé un 1 à l'extrême droite sur le LSB, puis nous l'avons décalé de 3 positions vers la gauche. Souvenez-vous : en cas de décalage, un 0 est toujours intercalé à la position précédente. Ce n'est pas ce que nous voulons ici ! Alors que faire ? La solution réside dans un autre opérateur qui appartient à la catégorie des opérateurs de bits. Il s'agit du OU binaire. Pour mieux comprendre l'effet de ces opérateurs, nous allons examiner le tableau de valeurs suivant :

Tableau 6-2 ▶
L'opérateur binaire OU

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

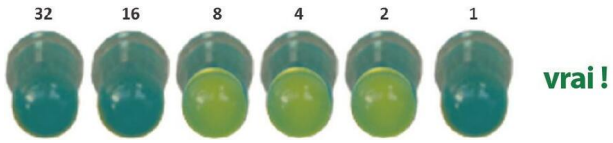
Les colonnes *A* et *B* contiennent des valeurs de départ logiques et la colonne *Q* présente le résultat de la liaison logique ou binaire OU. L'opérateur logique OU est symbolisé par la barre verticale | (pipe). Si nous modifions la ligne

```
PORTB = 1 << 3;
```

en

```
PORTB |= 1 << 3;
```

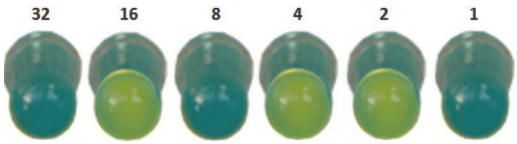
le bit correspondant est correctement placé, sans modifier pour autant les bits existants. Le résultat est le suivant :



Comment cela s'explique-t-il ? Quand une action de décalage progressive est constamment liée au masque de bits précédent sur le PORT B par un l'opérateur logique OU, alors tous les bits qui contiennent un 0 ne sont pas modifiés et tous ceux qui contiennent un 1 sont placés.

Supprimer des bits

Les bits placés peuvent bien sûr aussi être supprimés. Mettons que la combinaison de bits suivante ait été obtenue par la ligne de code ci-dessous.




```
PORTB = 0b0010110;
```

Nous voulons supprimer le bit ayant la valeur 2 à la position 1 afin d'éteindre la LED correspondante. L'état de toutes les autres LED ne doit pas s'en trouver modifié. Nous utiliserons un autre opérateur appartenant à la catégorie des opérateurs de bits : l'opérateur logique ET. Le tableau de valeurs ci-dessous en explique le fonctionnement.

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Tableau 6-3
L'opérateur binaire ET

Le résultat de cette liaison peut uniquement être un 1 lorsque les deux valeurs d'entrées sont des 1. L'opérateur logique ET est représenté par le et commercial &. Tous les bits qui ont la valeur 0 dans le masque de liaison sont supprimés. Ceux qui ont la valeur 1 ne sont pas concernés. Le code du sketch est le suivant :

```
void setup() {
  DDRB  = 0b11111111; // PORT B entièrement défini comme SORTIE
  PORTB = 0b00010110; // Masque de bits de départ
  delay(500);
  PORTB &= 0b11111101;
}

void loop() { /* vide */ }
```

La ligne

```
PORTB &= 0b11111101;
```

produit presque un masque. Les bits existants ne passent pas aux endroits où il y a un 0

Problèmes courants

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

- Vos fiches de raccordement sur la plaque correspondent-elles vraiment au circuit ?
- Pas de court-circuit éventuel entre elles ?
- Les LED ont-elles été mises dans le bon sens ? Autrement dit, la polarité est-elle correcte.

- Les résistances ont-elles bien les bonnes valeurs ?
- Le code du sketch est-il correct ?

Qu'avez-vous appris ?

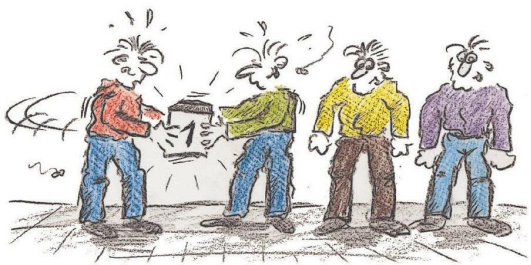
- Vous avez fait la connaissance d'une forme spéciale de variable vous permettant d'enregistrer plusieurs valeurs d'un même type de donnée. Elle est appelée tableau *(array)*. On accède à ses différents éléments au moyen d'un index.
- La boucle `for` vous permet d'exécuter plusieurs fois une ou plusieurs lignes de code. Elle est gérée par une variable de contrôle, active dans la boucle et initialisée avec une certaine valeur initiale. Une condition vous a permis de définir combien de fois la boucle doit s'exécuter. Vous contrôlez ainsi quel domaine de valeurs la variable traite.
- Vous pouvez réunir plusieurs instructions, qui sont ensuite toutes exécutées par exemple dans le cas d'une boucle `for`, en constituant un bloc au moyen de la paire d'accolades.
- La variable de contrôle, dont nous venons de parler, est utilisée pour modifier l'index d'un tableau et accéder ainsi à ses différents éléments.
- La manipulation de registre vous a permis de commander les broches numériques d'un port pour créer un séquenceur de lumière.

Extension de port

Nous avons vu dans le montage précédent comment programmer la commande des multiples LED d'un séquenceur de lumière. Votre carte Arduino ne disposant que d'un nombre limité de sorties numériques, ces précieuses ressources pourraient finir par vous manquer pour ajouter d'autres LED à votre séquenceur de lumière. Par ailleurs, vous voulez peut-être aussi connecter quelques capteurs sur des entrées numériques. Vous aurez donc encore moins de broches numériques à disposition. Comment résoudre ce problème ?

Le registre à décalage

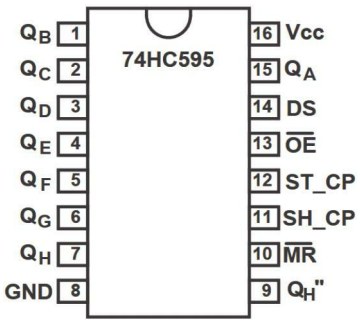
Il existe plusieurs solutions d'extension de port, en voici une. Je dois utiliser pour cela un *registre à décalage*. Vous vous demandez certainement ce que c'est et comment il opère. Dans cette expérimentation, un circuit intégré (*IC* pour *Integrated Circuit*) sera relié pour la première fois à votre carte Arduino. Un registre à décalage est un circuit géré par un signal d'horloge et doté de plusieurs sorties disposées l'une derrière l'autre. À chaque période d'horloge, le niveau présent à l'entrée du registre est transmis à la sortie suivante. Cette information passe ainsi par toutes les sorties existantes. L'illustration ci-dessous montre l'action zélée du registre de décalage décalant infatigablement le niveau logique `HIGH` à la position suivante.



◀ **Figure 7-1**
Attrape !

Le circuit intégré 74HC595, que nous utilisons ici, dispose d'une entrée série par laquelle les données sont entrées et de huit sorties équipées de registres mémoire internes pour conserver les états. Seules trois broches numériques sont nécessaires à l'alimentation, lesquelles fournissent des données au module qui, de son côté, commande ses huit sorties. C'est en soi une économie majeure, car le circuit 74HC595 peut être cascadé, permettant ainsi une expansion quasi illimitée des sorties numériques. De quoi s'agit-il exactement ? Voyons de plus près les différentes entrées et sorties de ce circuit. La figure suivante illustre le brochage du circuit, vu de dessus.

Figure 7-2 ▶
Brochage du registre à décalage 74HC595

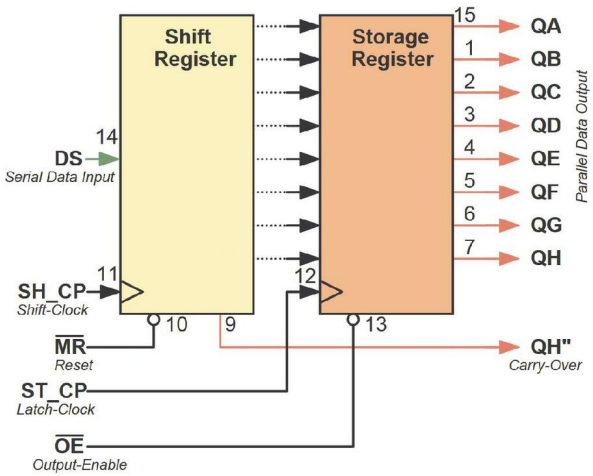


TRAIT HORIZONTAL AU-DESSUS D'UNE DÉSIGNATION DE BROCHE

Quand la désignation d'une broche comporte un trait horizontal, comme pour les broches 10 et 13 de la figure précédente, cela signifie que ce sont des entrées de signaux actives au niveau `LOW`. Pour un *master reset* (réinitialisation générale) sur la broche 10, la réinitialisation sera donc déclenchée en présence d'un niveau `LOW` et, en fonctionnement normal, elle devra être raccordée à une tension d'alimentation Vcc.

La figure suivante illustre le fonctionnement d'un registre à décalage avec un traitement en deux étapes.

Figure 7-3 ▶
Fonctionnement du registre à décalage 74HC595



1re étape

Le registre à décalage *(Shift Register)* sur le côté gauche reçoit les données en série sur la broche *DS* (14) et les enregistre temporairement. Quand le front est positif (bascule de 0 à 1) sur la broche *SH_CP* (11), toutes les valeurs du flux de données entrent une par une dans le registre interne.

2e étape

Les données doivent encore être transférées dans le registre mémoire *(Storage Register)* afin que les données auparavant série soient transmises en parallèle aux sorties Q_A à Q_H. Cela s'effectue sur un front positif (bascule de 0 à 1) sur la broche *ST_CP* (12). Pour que le transfert s'exécute correctement vers les sorties, la broche *OE (Output Enable)* (13) doit être reliée à la masse, car elle est active au niveau `LOW`. Le tableau suivant récapitule les différentes broches et leur signification.

Broche	Signification
Vcc	Tension d'alimentation + 5 V
GND	Masse 0 V
Q_A-Q_H	Sorties parallèles 1 à 8
Q_H"	Sortie série (entrée pour un deuxième registre à décalage)
MR	Master Reset (`actif LOW`)
SH_CP	Registre à décalage, entrée d'horloge (`Shiftregister clock input`)
ST_CP	Registre mémoire, entrée d'horloge (`Storageregister clock input`)
œ	Activation de la sortie (`Output enable/actif LOW`)
DS	Entrée série (`Serial data input`)

◀ Tableau 7-1
Signification des broches du registre à décalage 74HC595

Le mode de fonctionnement du registre à décalage peut se résumer ainsi : quand le niveau à l'entrée d'horloge *SH_CP* passe de `LOW` à `HIGH`, le niveau à l'entrée série DS est lu, transmis à l'un des registres internes et enregistré temporairement. Mais cela ne signifie pas pour autant transmis aux sorties Q_A à Q_H. C'est seulement une impulsion d'horloge à l'entrée *ST_CP* de `LOW` à `HIGH` qui fait transmettre toutes les informations des registres internes aux sorties. C'est utile, car ce n'est que quand toutes les informations ont été lues à l'entrée série qu'elles sont censées être détectées aux sorties. Le changement du niveau logique de `LOW` à `HIGH` est appelé *contrôle par front montant d'horloge*, car une action n'est entreprise que quand un changement de niveau se produit de la manière décrite.

Voyons maintenant un peu ce qui se passe dans le registre à décalage...

Voici justement *SH_CP* au travail. Quand il tourne la pancarte de `LOW` à `HIGH`, le candidat potentiel, qui se trouve dans la zone *DS-area*, passe dans le registre suivant et attend la suite de son voyage vers la sortie.

Figure 7-4 ▶
SH_CP préparant les données série

La figure suivante montre *ST_CP* en train de faire partir les données des registres internes vers les sorties.

Figure 7-5 ▶
ST_CP autorisant le départ des données des registres vers les sorties

Quand il tourne la pancarte de `LOW` à `HIGH`, les portes des registres internes s'ouvrent et alors seulement les données peuvent trouver le chemin de la sortie. Le procédé croqué ici sera reproduit dans plusieurs sketches pour que vous puissiez voir en direct comment marche le registre à décalage. Nous allons tout faire de A à Z, mais vous verrez à la fin qu'il existe une instruction bien commode pour toutes les actions à entreprendre l'une derrière l'autre, qui vous épargnera beaucoup de travail et vous facilitera les choses.

Registre à décalage conventionnel

Certains d'entre vous seront certainement intéressés par la structure interne d'un registre à décalage. Les différents modèles intègrent des composants variés. Nous allons examiner un exemple rudimentaire : si l'on raccorde successivement plusieurs bascules ou verrous (*flipflops*, en anglais), c'est-à-dire des circuits pouvant accepter deux états stables, on obtient un registre à décalage. Le schéma suivant représente un registre à décalage réalisé avec 4 verrous D.

Vous trouverez de plus amples informations sur le verrou D à l'adresse suivante :

https://fr.wikipedia.org/wiki/Bascule_(circuit_logique)

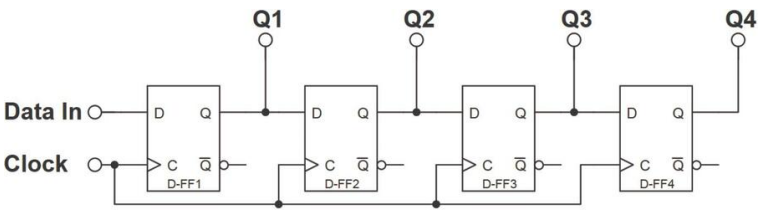


Figure 7-6
Registre à décalage réalisé avec 4 verrous D

Les niveaux sont lus aux entrées via *Data In*, puis ils poursuivent leur route à chaque période d'horloge *Clock*. Les sorties *Q1* à *Q4* affichent de façon parallèle les informations arrivées en série.

Composants nécessaires

Ce montage nécessite les composants suivants.

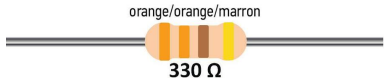
Composant	
1 registre à décalage 74HC595	
8 LED rouges	
8 résistances de 330 Ω	
1 résistance de 10 kΩ	
1 bouton-poussoir miniature	

Tableau 7-2
Liste des composants

Schéma

Le schéma montre les différentes LED avec leurs résistances série de 330 ohms, qui sont commandées par le registre à décalage 74HC595. L'entrée *master reset* de la puce est connectée, à travers la résistance pull-up, à la tension d'alimentation +5 V, si bien que le reset ne se déclenche pas tant que le bouton-poussoir n'est pas enfoncé puisque l'entrée *MR* est active au niveau `LOW`. On note la présence d'un trait horizontal au-dessus de *MR*, qui correspond à une négation. L'entrée *output enabled* est également active au niveau `LOW` et reliée par un fil à la masse, car les sorties doivent être toujours actives. Le registre à décalage est commandé par les broches Arduino 8, 9 et 10 avec les fonctions décrites précédemment.

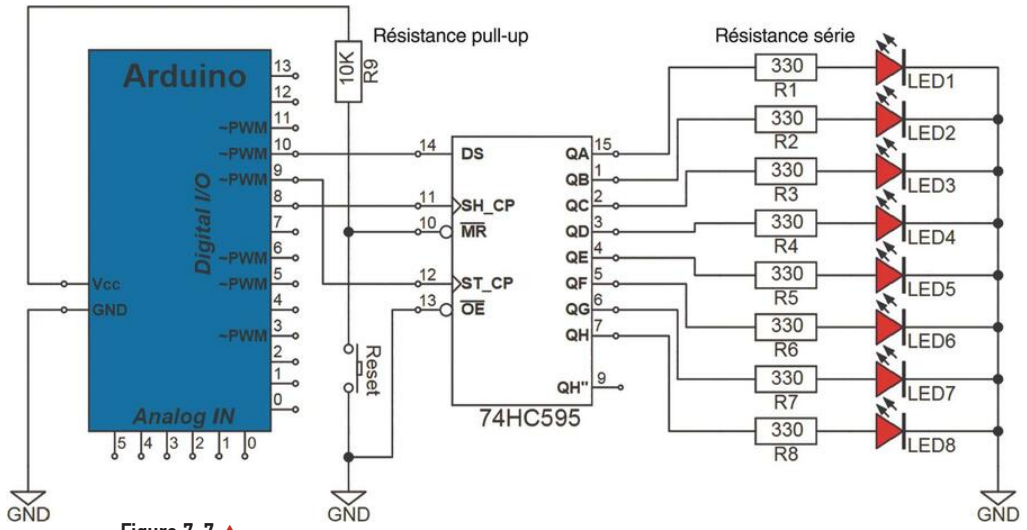
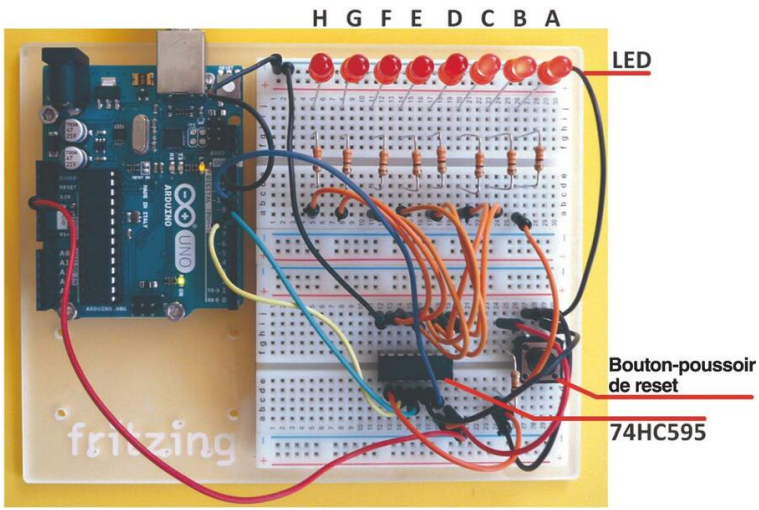


Figure 7-7 ▲
Carte Arduino commandant le registre à décalage 74HC595 par trois lignes de signaux

Si vous lancez le sketch, que nous verrons bientôt, la première LED s'allume immédiatement sur la sortie Q_A, car vous avez entré une seule fois `1` dans le registre à décalage. Vous devez actionner non seulement le bouton-poussoir du circuit, mais aussi le bouton de reset de la carte Arduino pour effectuer un reset (réinitialisation).

Réalisation du circuit

Vérifiez que les raccordements ont été correctement effectués sur la plaque d'essais qui se remplit de plus en plus. Restez concentré !



◀ Figure 7-8
Réalisation du circuit avec le registre à décalage 74HC595

Sketch Arduino

Voici le code du sketch pour commander le registre à décalage 74HC595 au moyen de trois lignes de sorties numériques. Les broches suivantes sont nécessaires sur le registre :

- *SH_CP* (registre à décalage, entrée d'horloge) ;
- *ST_CP* (registre mémoire, entrée d'horloge) ;
- *DS* (entrée série pour les données).

Des variables sont affectées aux trois lignes de données, variables auxquelles j'ai donné les noms suivants :

- SH_CP est `shiftPin` ;
- ST_CP est `storagePin` ;
- DS est `dataPin`.

Ce sketch met l'entrée série *DS* sur `HIGH`, niveau qui est ensuite transféré dans le registre interne quand l'entrée d'horloge *SH_CP* du registre à décalage passe de `LOW` à `HIGH`. Les sorties sont alors programmées et enregistrées via les registres internes au moyen de l'entrée d'horloge *ST_CP* du registre mémoire.

```
int shiftPin = 8;     // SH_CP
int storagePin = 9;  // ST_CP
int dataPin = 10;    // DS

// Réinitialisation de toutes les broches → niveau LOW
void resetPins() {
  digitalWrite(shiftPin, LOW);
  digitalWrite(storagePin, LOW);
  digitalWrite(dataPin, LOW);
}

void setup() {
  pinMode(shiftPin, OUTPUT);
  pinMode(storagePin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  resetPins(); // Mise de toutes les broches sur LOW
  // Mise de DS sur HIGH pour reprise ultérieure par SH_CP
  digitalWrite(dataPin, HIGH); // DS
  delay(20); // Brève pause avant traitement
  // Transmission du niveau à DS dans registres mémoire internes
  digitalWrite(shiftPin, HIGH); // SH_CP
  delay(20); // Brève pause avant traitement
  //Transmission des registres mémoire internes aux sorties
  digitalWrite(storagePin, HIGH); // ST_CP
  delay(20);
}

void loop(){ /* vide */ }
```

Comme je l'ai mentionné plus haut, seule la sortie Q_A s'allume après la transmission du sketch, ce qui n'a rien de spectaculaire. Mais cela ne va pas tarder à devenir plus intéressant. Promis !

Revue de code

Les variables sont d'abord pourvues des informations de broche nécessaires puis toutes les broches sont programmées en tant que sorties au début de la fonction `setup`. Vous rencontrez pour la première fois dans ce montage une fonction écrite par vous-même. Une fonction n'a en soi rien de nouveau pour vous puisque `setup` et `loop` font déjà partie de cette catégorie de structures logicielles. Je souhaite cependant revenir sur le sujet pour en préciser le sens. Une fonction peut être considérée comme une sorte de sous-programme qui peut toujours être appelé dans le déroulement normal d'un sketch. Elle est invoquée par son nom et peut tout aussi bien retourner une valeur à l'appelant qu'enregistrer plusieurs valeurs transférées nécessaires au calcul ou au traitement. La structure formelle d'une fonction est la suivante :

```
Type de donnée de retour Nom (paramètre)
{
   //Une ou plusieurs instructions
}
```

◀ **Figure 7-9**
Structure de base d'une fonction

La partie entourée est appelée *signature de la fonction* et représente l'interface formelle avec la fonction. Cette dernière est comparable à une boîte noire : vous n'avez pas besoin de savoir comment elle fonctionne. Il vous suffit de connaître la structure de l'interface et de savoir sous quelle forme une valeur est retournée. Vous programmez ici bien entendu la fonction elle-même et devez pour le moins connaître la logique qu'elle renferme. Certaines fonctions peuvent également être obtenues par exemple sur Internet, dans la mesure où leur usage n'est pas limité techniquement par une licence, et utilisées dans votre montage. Peu importe de savoir comment elles fonctionnent du moment qu'elles ont été programmées et testées avec succès par d'autres. Le principal est qu'elles fonctionnent ! Mais revenons à notre définition de la fonction. Si elle renvoie une valeur en retour à l'appelant, comme le fait par exemple `digitalRead`, vous devez indiquer le type de donnée en question dans votre fonction.

Supposons que vous vouliez retourner des valeurs qui sont toutes des nombres entiers, le type de donnée est alors `Integer` défini par le mot-clé `int`. Si toutefois aucun retour n'est requis, vous devez le faire savoir par le mot-clé `void` (qui signifie : *vide*) qui précède déjà les deux fonctions principales `setup` et `loop`.

Nous avons dit que les fonctions sont toujours invoquées par leur nom. Mais qu'en est-il des deux fonctions `setup` et `loop` ? Pas besoin de stipuler quelque part dans le code qu'elles doivent être appelées et pourtant ça marche. Comment est-ce possible ?

En fait, `setup` et `loop` sont des *fonctions systémiques* qui sont appelées implicitement. Comme vous l'avez remarqué, vous n'avez pas besoin de vous en occuper. Si cela vous intéresse, vous trouverez dans le répertoire d'installation sous

C:\Program Files (x86)\Arduino\hardware\arduino\avr\cores\arduino

le fichier `main.cpp` que vous pourrez ouvrir avec un éditeur de texte. Vous verrez alors ce qui suit :

```
33  int main(void)
34  {
35      init();
36
37      initVariant();
38
39  #if defined(USBCON)
40      USBDevice.attach();
41  #endif
42
43      setup();
44
45      for (;;) {
46          loop();
47          if (serialEventRun) serialEventRun();
48      }
49
50      return 0;
51  }
```

La fonction directement appelée en début de programme avec C++ est nommée `main`, comme c'est le cas ici. Elle sert quasiment de point d'accès, pour que le programme sache par quoi il doit commencer. `main` contient plusieurs appels de fonction qui sont traités l'un après l'autre. On y trouve entre autres la fonction `setup` et l'appel de la fonction `loop` dans une boucle sans fin définie par `for(;;)`. Vous reconnaissez certainement les déroulements ou plutôt les relations qui se créent en coulisses au début d'un sketch quand il s'agit d'appeler `setup` ou `loop`.

Si une ou plusieurs variables sont à fournir à votre fonction, celles-ci sont indiquées entre parenthèses derrière son nom, séparées par des virgules, avec leur type de donnée correspondant. Les parenthèses sont nécessaires même s'il n'y a rien entre elles, faute de variables. La signature est suivie du corps de fonction, formé par la paire d'accolades. Toutes les instructions, qui se trouvent entre ces deux accolades, font partie de la fonction et sont traitées séquentiellement de haut en bas lors de l'appel. Mais revenons au code. En quoi est-ce utile d'écrire une fonction particulière ? Très simple ! Ça l'est toujours quand les mêmes instructions sont à exécuter plusieurs fois dans le code et c'est ici le cas. Je dois exécuter la suite d'instructions

```
digitalWrite(shiftPin, LOW);
digitalWrite(storagePin, LOW);
digitalWrite(dataPin, LOW);
```

à divers endroits pour réinitialiser – autrement dit, pour remettre sur `LOW` – les niveaux sur les différentes broches numériques. Sans fonction, le sketch compterait un grand nombre de lignes de code en plus et manquerait donc de clarté.

CODE REDONDANT

Le code source, qui revient plusieurs fois avec la même séquence d'instructions dans le sketch, est appelé *code redondant* ou *redondance de code*. Le mieux est de le stocker dans une fonction à laquelle vous donnez un nom suffisamment évocateur pour en saisir le sens. Si vous devez procéder à une modification, vous intervenez de manière centrale au sein de la fonction et non pas un peu partout dans le code, ce qui est générateur de bien des erreurs et très chronophage.

Au début du sketch, l'appel de fonction :

```
resetPins(); // Mise de toutes les broches sur LOW
```

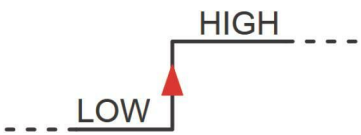
permet de mettre les broches 8, 9 et 10 au niveau `LOW`. Le premier signal de niveau `HIGH` est ensuite appliqué à *DS* par la ligne :

```
digitalWrite(dataPin, HIGH); // DS
```

Puis une attente de 20 ms s'écoule avant que la ligne

```
digitalWrite(shiftPin, HIGH); // SH_CP
```

ne transmette le niveau `HIGH` de *DS* au registre mémoire interne. Il faut ici tenir compte du fait que ce n'est possible qu'au moyen d'un contrôle par front montant de `LOW` vers `HIGH`.



Il n'y a pas encore de transfert en direction du port de sortie. Une nouvelle attente de 20 ms s'écoule, et enfin la ligne

```
digitalWrite(storagePin, HIGH); // ST_CP
```

déclenche la transmission des registres mémoire internes aux sorties, ce qui revient à commander les LED dans le cas présent. Un changement de niveau de `LOW` à `HIGH` est ici aussi nécessaire, d'où le recours préalable à la fonction `resetPin` qui permettra plus tard de changer encore le niveau de `LOW` à `HIGH`.

Extension du sketch : première partie

Complétons maintenant un peu le sketch de telle sorte que vous puissiez rentrer plusieurs valeurs dans l'entrée série. Ce n'est encore qu'un degré intermédiaire et non pas la solution finale que je souhaite vous présenter. Ce code doit transmettre au registre à décalage une séquence stockée dans un tableau de données. La construction du circuit reste la même.

```
int shiftPin = 8;     // SH_CP
int storagePin = 9;  // ST_CP
int dataPin = 10;    // DS
int dataArray[] = {1, 0, 1, 0, 1, 1, 0, 1};

void resetPins() {
  digitalWrite(shiftPin, LOW);
  digitalWrite(storagePin, LOW);
  digitalWrite(dataPin, LOW);
}

void putPins(int data[]){
  for(int i = 0; i < 8; i++) {
    resetPins();
    digitalWrite(dataPin, data[i]); delay(20);
    digitalWrite(shiftPin, HIGH); delay(20);
  }
}

void setup() {
  pinMode(shiftPin, OUTPUT);
  pinMode(storagePin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  resetPins();          // Mettre toutes les broches à LOW
  putPins(dataArray); // Régler les broches sur le tableau de données
  // Transmission des registres mémoire internes aux sorties
  digitalWrite(storagePin, HIGH); // ST_CP
}

void loop() { /* vide */ }
```

Voyons maintenant comment le code accomplit son travail. Tout tourne autour du tableau de données qui contient le modèle de commande des différentes LED. Il s'agit donc de la ligne de déclaration et d'initialisation suivante :

```
int dataArray[] = {1, 0, 1, 0, 1, 1, 0, 1};
```

Le code lit les éléments du tableau de gauche à droite et rentre les valeurs dans le registre à décalage. Un `1` signifie une LED allumée et un `0` une LED éteinte.

Vous remarquerez que nous avons utilisé les valeurs 1 et 0 pour commander les LED. Ça marche ? Ne vaut-il pas mieux travailler avec les noms de constante `HIGH` et `LOW` ?

J'ai préféré utiliser les valeurs `1` et `0` parce que ce sont elles précisément qui se cachent derrière les constantes `HIGH` et `LOW`. Normalement, je ne suis pas pour les *magic numbers,* mais il m'a semblé que dans ce cas, je pouvais faire une exception. `1` et `0` étant aussi les valeurs logiques, vous ne devriez pas avoir trop de mal à comprendre ! Vous pouvez bien entendu écrire à la place de :

```
int dataArray[] = {1, 0, 1, 0, 1, 1, 0, 1};
```

la ligne suivante :

```
int dataArray[] = {HIGH, LOW, HIGH, LOW, HIGH, HIGH, LOW, HIGH};
```

Mais revenons au code et à sa manière d'exploiter le tableau. La chose n'est pas si simple, car j'ai ajouté une fonction nommée `putPins`, qui a pour tâche de remplir le registre à décalage. Elle présente un paramètre de transmission capable d'enregistrer non pas une variable normale, mais tout un tableau de données. Il suffit de transmettre le tableau de données comme argument dans la fonction :

```
putPins(dataArray);
```

La fonction est définie comme suit :

```
void putPins(int data[]){
  for(int i = 0; i < 8; i++) {
    resetPins(); // Remise à zéro des broches et préparation à la commande
                 // par front montant
    digitalWrite(dataPin, data[i]); delay(20);
    digitalWrite(shiftPin, HIGH); delay(20);
  }
}
```

On peut voir qu'un tableau du type de donnée `int` a été déclaré avec deux crochets dans la signature de la fonction. Au moment où la fonction est appelée, le tableau initial `dataArray` est copié dans `data`, qui est ensuite exploité dans la fonction. Puis chaque élément du tableau est envoyé dans l'entrée série, au moyen de la boucle `for` (que vous connaissez déjà), via :

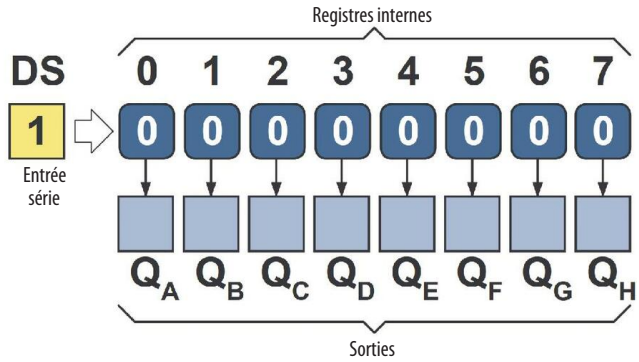
```
digitalWrite(dataPin, data[i]);
```

et placé lors de l'étape suivante dans le premier registre interne via :

```
digitalWrite(shiftPin, HIGH);
```

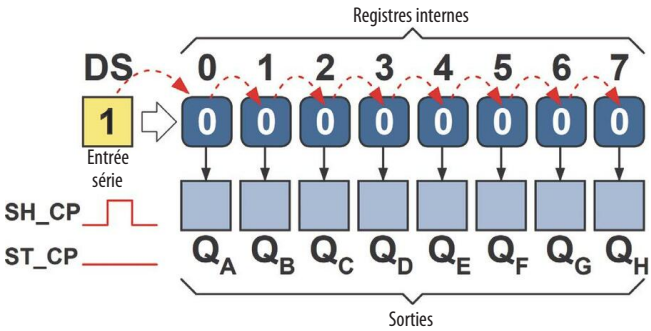
Le tout s'effectue huit fois (de `0` à `7`), chaque registre interne transmettant ses valeurs au suivant. Les figures suivantes montrent les choses encore plus clairement.

Figure 7-10 ▶
Registre à décalage (phase 1)

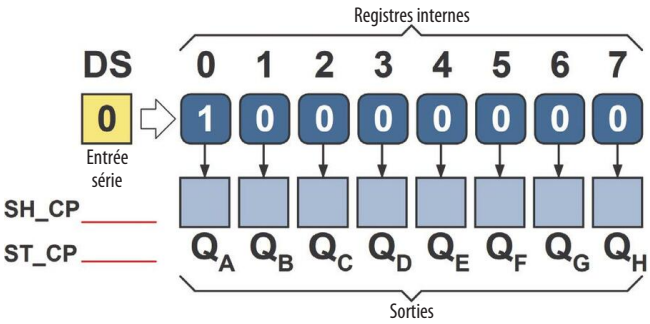


Au début, les registres sont encore tous vides. Un bit `1` attend toutefois déjà à l'entrée d'être transféré dans le premier registre interne sur le côté gauche.

Figure 7-11 ▶
Registre à décalage (phase 2)

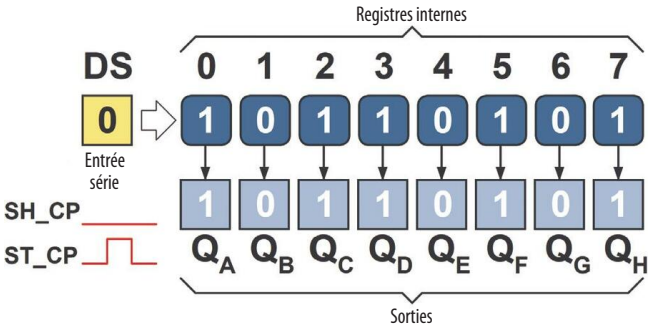


Le `1` qui se trouve à l'entrée série est entré, pendant le front montant de *SH_CP*, dans le premier registre interne. Les contenus de tous les registres sont décalés d'une position vers la droite. Cette action donne les états illustrés sur la page suivante.



◀ Figure 7-12
Registre à décalage (phase 3)

À l'entrée se trouve maintenant un `0` qui sera lui aussi entré à la prochaine impulsion de *SH_CP* dans le premier registre interne. Mais avant, l'état du septième registre interne sera passé au huitième, le sixième au septième, etc. Enfin, le 0 est entré dans le premier registre. Passons directement au moment où toutes les valeurs du tableau ont été entrées, conformément au schéma précédent, dans les registres internes et où l'impulsion *ST_CP* a recopié les registres vers les sorties.



◀ Figure 7-13
Registre à décalage (phase 4)

Les valeurs du tableau lu sont appliquées aux sorties seulement maintenant, la première valeur entrée se trouvant complètement à droite et la dernière complètement à gauche.

Vous vous demandez peut-être comment faire pour inverser le comportement ? En effet, vous voudriez que la première valeur du tableau se trouve complètement à gauche et que la dernière complètement à droite se trouve à la sortie, de telle sorte que l'ordre soit quasiment inversé.

Pas de problème car où les broches ont-elles été déterminées ? Exact, dans la fonction `putPins` ! C'est la boucle `for` qui fixe l'ordre des différentes broches. Celui-ci sera inversé si vous appelez la dernière valeur à la place de

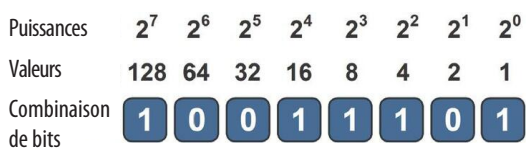
la première et la transmettez au registre à décalage. Voici le code modifié de la boucle `for` :

```
for(int i = 7; i >= 0; i--){
  // ...
}
```

Extension du sketch : deuxième partie

Maintenant que vous en savez assez sur le registre à décalage 74HC595, je voudrais vous présenter une instruction spéciale qui vous épargnera du travail. `shiftOut` est vraiment facile à utiliser. Mais je dois auparavant vous livrer quelques informations sur la sauvegarde des données dans l'ordinateur, informations qui sont vraiment importantes pour comprendre le fonctionnement d'un microcontrôleur. Je me sers pour mes réalisations du type de donnée `byte`, dont la taille est de 8 bits et qui peut stocker des valeurs comprises entre `0` et `255`. La figure suivante montre la valeur décimale 157 sous sa forme binaire 10011101.

Figure 7-14 ▶
Combinaison binaire pour le nombre entier 157



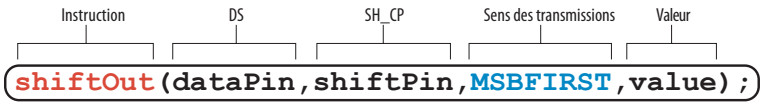
Si on regarde les puissances de plus près, on voit que la base est le chiffre 2. Nous autres humains comptons en base 10 en raison des dix doigts de nos mains. Les valeurs des différents chiffres d'un nombre dont donc 10^0, 10^1, 10^2, etc. Pour le nombre 157, cela donne $7 \times 10^0 + 5 \times 10^1 + 1 \times 10^2$ qui font bien sûr 157. Le microcontrôleur ne pouvant cependant stocker que deux états (HIGH et LOW), le système binaire (du latin *binarius*, deux chacun) est fondé sur la base 2. La valeur décimale de ladite combinaison binaire se calcule par conséquent comme suit, en commençant en principe par la valeur – ou bit – la plus petite :

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

NOTATION

La base figure derrière le nombre pour plus de clarté quand différents systèmes de numération sont utilisés.

Avec un nombre de 8 bits (ou 1 octet), vous pouvez représenter 256 valeurs distinctes (de 0 à 255). Ceci dit, revenons à l'instruction `shiftOut`, qui possède différents paramètres dont nous allons faire le tour.

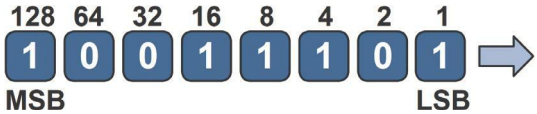


◀ **Figure 7-15**
Instruction `shiftOut` avec ses nombreux arguments

Les arguments `dataPin`, `shiftPin` ou encore la valeur à transmettre sont censés être clairs. Mais que signifie la constante `MSBFIRST` ? Cet argument permet de définir le sens de transmission des bits. Dans le cas d'un octet, le bit de poids fort est nommé *Most Significant Bit* (MSB) et le bit de poids faible *Least Significant Bit* (LSB). Vous pouvez ainsi définir quel bit doit être transféré en premier dans le registre à décalage.

MSBFIRST

LSBFIRST



Voici le code complet avec l'instruction `shiftOut`. Le circuit n'a pas non plus besoin ici d'être modifié.

```
int shiftPin = 8;   // SH_CP
int storagePin = 9; // ST_CP
int dataPin = 10;   // DS
byte value = 157;   // Valeur à transférer

void setup() {
  pinMode(shiftPin, OUTPUT);
  pinMode(storagePin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  digitalWrite(storagePin, LOW);
  shiftOut(dataPin, shiftPin, MSBFIRST, value);
  digitalWrite(storagePin, HIGH);
  delay(20);
}
```

INDICATION D'UNE VALEUR BINAIRE

Vous pouvez saisir directement la combinaison binaire au lieu du nombre décimal 157 lors de l'initialisation des variables et éviter ainsi la conversion. Tapez seulement `B10011101`. Le préfixe `B` indique qu'il s'agit d'une combinaison binaire avec laquelle la variable doit être initialisée.

Ce chronogramme vous montre les niveaux des trois lignes de données, les unes par rapport aux autres, pour commander le registre à décalage pendant le déroulement chronologique.

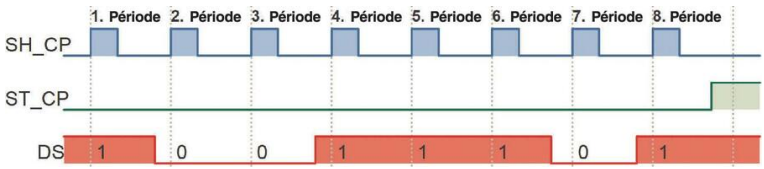


Figure 7-16 ▶ Chronogramme pour le nombre transféré `157` (`B10011101`)

On voit tout en haut le signal d'horloge *SH_CP* pour la prise en charge des données à l'entrée série DS. À l'issue de la 8e période, le niveau de *ST_CP* passe de `LOW` à `HIGH` et les données des registres internes sont transmises aux sorties. Essayez avec différentes valeurs et différents sens de transmission pour bien comprendre.

POUR ALLER PLUS LOIN

Pour compléter ce chapitre, vous pouvez effectuer une recherche sur Internet sur les mots-clés :

- 74HC595
- 75HC595 fiche technique ;
- 74HC595 datasheet.

Problèmes courants

Si les LED ne s'allument pas l'une après l'autre, 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 ?
- Pas de court-circuit éventuel ?
- Les différentes LED sont-elles correctement branchées ? La polarité est-elle correcte ?
- Les résistances ont-elles bien les bonnes valeurs ?

- Le registre à décalage est-il correctement câblé ? Contrôlez encore une fois tous les raccordements qui sont nombreux.
- Le code du sketch est-il correct ?

Qu'avez-vous appris ?

- Vous savez ce qu'est un registre à décalage de type 74HC595 avec une entrée série et huit sorties.
- Le premier sketch permet de commander les trois lignes de données *SH_CP, ST_CP et DS,* les signaux d'horloge étant contrôlés par un front d'horloge montant, autrement dit ne réagissant que quand le niveau passe de `LOW` à `HIGH`.
- L'instruction `shiftOut` permet d'envoyer au registre à décalage des combinaisons de bits au moyen de nombres non seulement décimaux, mais aussi binaires.
- Vous pouvez initialiser une variable du type de donnée `byte` avec un nombre entier, par exemple `157`, ou à l'aide de la combinaison de bits correspondante qui doit être précédée du préfixe `B`, soit par exemple `B10011101`.

8 Comment créer une bibliothèque ?

Le jour viendra où vos connaissances vous permettront de réaliser vos propres idées, que d'autres n'auront peut-être pas encore eues. Mais peut-être souhaiterez-vous aussi améliorer un projet existant parce que votre solution est plus élégante et moins compliquée à transposer. D'innombrables développeurs de logiciels se sont penchés avant vous sur les questions les plus diverses et ont programmé des bibliothèques pour épargner du travail et du temps aux autres développeurs. À travers ce projet, nous allons découvrir les grands principes de ces bibliothèques et leur création. Si le langage de programmation C++ – y compris la programmation orientée objet – vous a toujours intéressé, vous allez être servi !

Les bibliothèques

Une fois l'environnement de développement Arduino installé ou plutôt décompressé, vous disposez de quelques bibliothèques maison prêtes à l'emploi, appelées également librairies (*libraries* en anglais). Elles traitent de thèmes intéressants, tels que commander :

- un servomoteur ;
- un moteur pas-à-pas ;
- une connexion Wi-Fi.

Ces bibliothèques sont stockées dans le répertoire `libraries` du répertoire d'installation d'Arduino. Vous pouvez utiliser l'Explorateur Windows ou passer par l'environnement de développement Arduino pour savoir quelles sont les bibliothèques disponibles. On y trouve une entrée de menu spéciale `Sketch|Import Library` permettant d'afficher la liste correspondante.

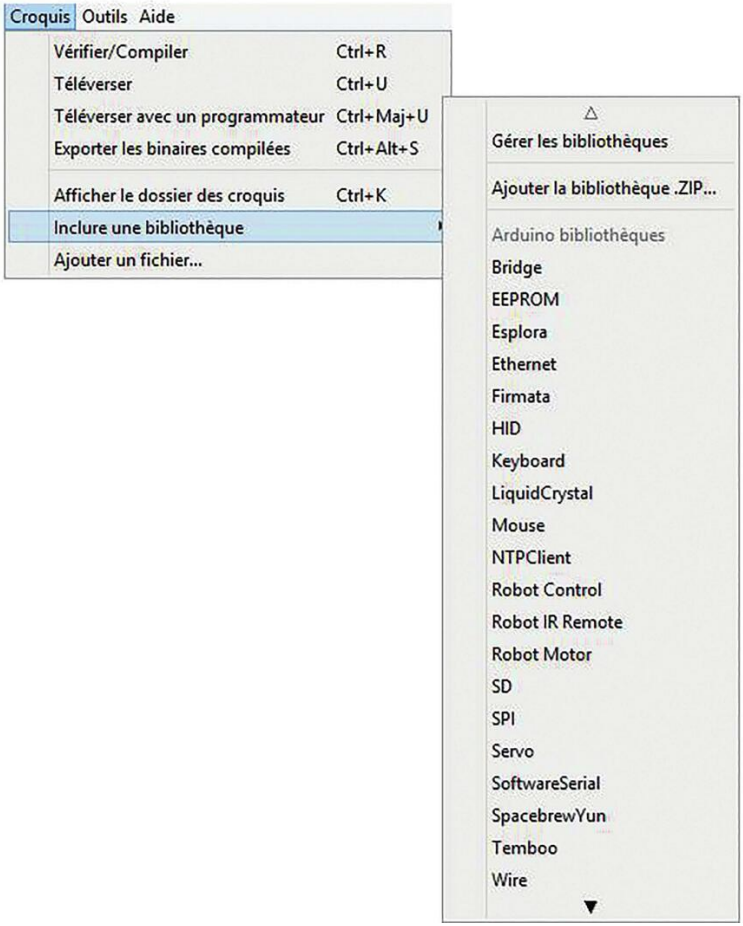
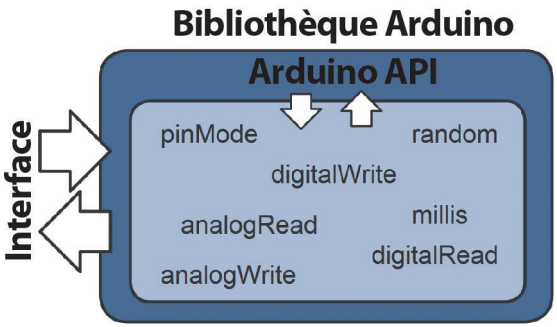


Figure 8-1 ▶
Afficher ou importer des bibliothèques

Les entrées du menu coïncident avec les répertoires du dossier `libraries`. Tout cela est bien beau, mais voyons d'abord comment une bibliothèque Arduino fonctionne et ce que nous pouvons faire avec.

Qu'est-ce qu'une bibliothèque exactement ?

Avant de passer à un exemple concret, vous devez savoir ce qu'est une bibliothèque. J'ai dit déjà qu'elle servait quasiment à empaqueter et réunir des tâches de programmation plus ou moins complexes en un paquet de programme. La figure 8-2 montre la coopération entre une bibliothèque Arduino et l'API Arduino.



◀ Figure 8-2
Comme fonctionne une bibliothèque Arduino ?

Nous avons affaire à deux couches de programme interdépendantes. Je procède de l'intérieur vers l'extérieur. J'ai appelé la couche interne API Arduino. (API est l'abréviation d'*Application Programming Interface* c'est une interface vers toutes les instructions Arduino disponibles.) Je n'en ai sélectionné que très peu par manque de place. La couche externe est constituée par la bibliothèque Arduino, qui enveloppe la couche interne. Elle est de ce fait appelée *wrapper* (enveloppe) et se sert de l'API Arduino. Pour pouvoir accéder à la couche wrapper, une interface doit être mise en œuvre, car vous entendez bien sûr exploiter la fonctionnalité d'une bibliothèque. Une interface est un portail d'accès à l'intérieur de la bibliothèque, qui est en soi une unité fermée. Le terme technique est encapsulation. Vous allez bientôt voir en détail de quoi il s'agit et en quoi cela concerne le langage de programmation C++.

En quoi les bibliothèques sont-elles utiles ?

Question idiote à laquelle j'ai déjà répondu plusieurs fois. Aussi me contenterai-je ici de vous en rappeler les avantages.

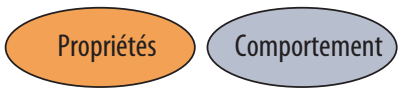
- Pour ne pas avoir à « réinventer la roue » chaque fois, les développeurs ont trouvé un moyen de stocker le code de programme dans une bibliothèque. Beaucoup de programmeurs dans le monde profitent de ces structures logicielles, qu'ils peuvent utiliser sans problème dans leurs propres projets. Le mot-clé est ici *réutilisation*.
- Une fois testée et débarrassée de ses erreurs, une bibliothèque peut être utilisée sans en connaître les déroulements internes. Sa fonctionnalité est encapsulée et cachée du monde extérieur. Il suffit au programmeur de savoir utiliser son *interface*.
- Son code propre en est d'autant plus clair et plus stable.

Que signifie programmation orientée objet ?

La programmation orientée objet (ou POO) est du chinois pour la plupart des débutants, et peut même réserver maux de tête et nuits blanches à certains. Mais ce n'est pas obligé et j'espère pouvoir y contribuer, je veux dire à votre compréhension et non à vos maux de tête ! Dans le langage de programmation C++, tout est considéré comme objet et ce style – ou paradigme – de programmation s'oriente vers la réalité qui nous entoure. Nous sommes cernés d'innombrables objets qui sont plus ou moins réels et que nous pouvons toucher et observer. Si vous regardez un objet banal de plus près, vous pourrez constater certaines propriétés. Prenons par exemple un dé pour ne pas sortir du sujet. Vous savez déjà comment programmer et construire un dé électronique. Vous avez sûrement, dans l'un de vos jeux de société, un dé quelconque que vous pouvez regarder de plus près. Que pourriez-vous en dire, si vous deviez le décrire le plus précisément possible à un extra-terrestre ?

- À quoi ressemble-t-il ?
- Quelle taille a-t-il ?
- Est-il léger ou lourd ?
- De quelle couleur est-il ?
- A-t-il des points ou des symboles ?
- Quel nombre ou symbole est sorti ?
- Que peut-on faire avec ? (question idiote, non ?)

Les éléments de cette liste peuvent être répartis en deux catégories.



Mais quel élément fait partie de quelle catégorie ?

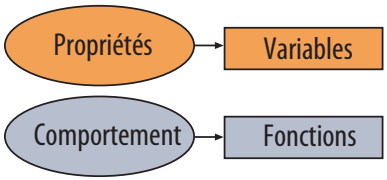
Tableau 8-1 ▶ Distinction entre propriétés et comportement

Propriétés	Comportement
Taille	Lancer le dé
Poids	
Couleur	
Points ou symboles	
Nombre ou symbole sorti	

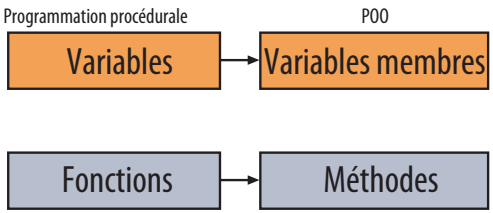
Seuls deux éléments de la liste sont pertinents pour la programmation prévue. Les autres sont certes intéressants, mais sans objet pour un dé électronique. Ces deux éléments sont :

- le nombre de points sorti (état) ;
- lancer le dé (action).

Voyons maintenant comment on charge des propriétés ou un comportement dans un sketch. Regardons la figure suivante.



Les propriétés sont consignées dans des variables et le comportement est géré par des fonctions. Mais dans le contexte de la programmation orientée objet, variables et fonctions ont une autre désignation. Pas de quoi paniquer cependant puisque c'est en définitive la même chose.

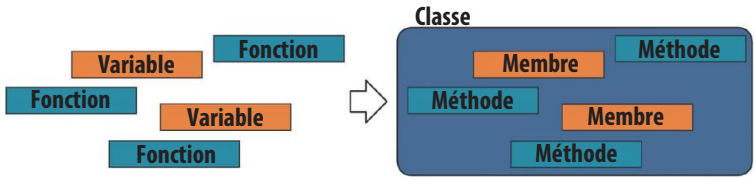


Les variables deviennent des variables *membres* (en anglais, *fields*) et les fonctions des *méthodes* (en anglais, *methods*).

Quelle avancée formidable, vous direz-vous ! Il suffit de rebaptiser deux éléments d'un programme pour avoir un nouveau *paradigme de programmation*. Le progrès tient à peu de choses, non ?

Dans la programmation procédurale que nous connaissons à travers les langages C ou Pascal, des instructions ayant un rapport logique, nécessaires pour résoudre un problème, sont rassemblées dans ce qu'on appelle des procédures semblables à nos fonctions. Les fonctions opèrent en principe au mieux avec les variables qui leur ont été transmises comme arguments ou, dans le cas défavorable, avec des variables globales qui ont été déclarées au début d'un programme. Celles-ci sont visibles dans tout le programme et chacun peut les modifier à sa convenance. Toutefois, cela comporte certains risques et c'est actuellement, tout bien pesé, la plus mauvaise variante pour traiter des variables ou des données. Variables et fonctions

ne forment aucune unité logique et vivent quasiment les unes à côté des autres dans le code, sans avoir aucun rapport direct entre elles. La programmation orientée objet comporte une structure appelée *classe*. Elle sert de container pour des variables membres ou des méthodes.

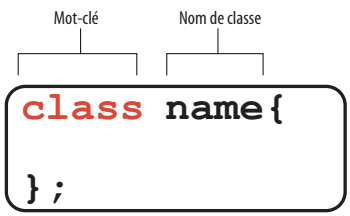


La classe enveloppe ses membres, appelés *members* dans la POO, à la manière d'un grand manteau. On ne peut en principe accéder aux membres qu'en passant par la classe.

Construction d'une classe

Qu'est-ce qu'une classe ? Si vous n'avez jamais eu affaire aux langages de programmation C++, Java et même C# pour ne citer que ceux-là, le terme ne vous en dira pas plus qu'un caractère chinois pour moi. Mais la chose est en fait assez facile à comprendre. Si vous regardez encore une fois le dernier graphique, vous verrez qu'une classe a vocation d'entourer et ressemble en quelque sorte à un container. Une classe est définie par le mot-clé `class`, suivi du nom qu'on lui a donné. Suit une paire d'accolades, que vous avez pu voir dans d'autres structures comme une boucle `for` et qui amène la formation d'un bloc. L'accolade finale est suivie d'un point-virgule.

Figure 8-3 ▶ Définition générale d'une classe

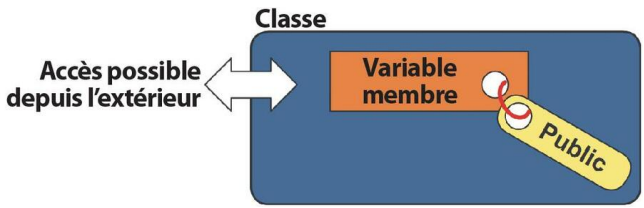


Comme je vous l'ai déjà dit, la classe est composée de différents membres sous forme de variables membres et de méthodes, qui se fondent, selon la définition de cette classe, en une unité. La POO offre diverses possibilités de réglementer l'accès aux membres.

Vous vous demandez peut-être à quoi sert cette réglementation. Quand vous définissez une variable ou plutôt une *variable membre* dans une classe, vous voulez pouvoir y accéder n'importe quand. Alors à quoi cela sert-il si vous ne pouvez plus ensuite accéder à la classe ?

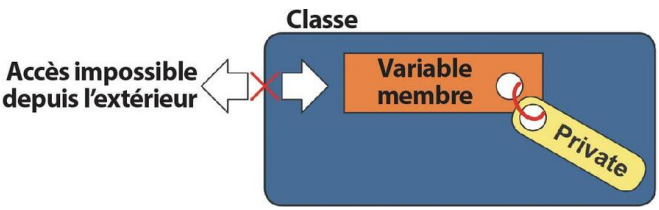
Vous avez bien compris le principe, appelé d'ailleurs *encapsulation*. On peut protéger certains membres contre le monde extérieur, de telle sorte qu'ils ne soient pas directement accessibles depuis l'extérieur de la classe. Le mot *directement* est ici important. Il existe bien sûr des possibilités d'y accéder. Ce sont les méthodes qui, par exemple, s'en chargent. Mais vous devez vous demander quel est le sens de tout cela.

En fait, on peut donc toujours influer directement sur les variables membres. Mais je pense que les figures suivantes vous permettront de mieux comprendre le principe.



◀ Figure 8-4
Accès à une variable membre de la classe

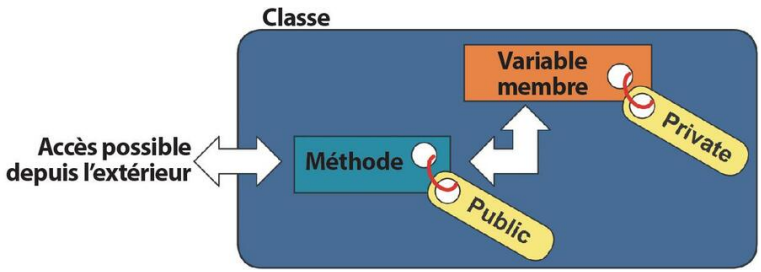
L'accès à la variable membre de la classe depuis l'extérieur est ici autorisé, car elle a reçu une certaine étiquette appelée *modificateur d'accès*. Elle a pour nom ici `public` et signifie à peu près ceci : l'accès est autorisé au public et tout un chacun peut s'en servir à sa guise. Imaginez maintenant le scénario suivant : une variable membre doit piloter un moteur pas-à-pas, la valeur indiquant l'angle. Seuls des angles compris entre 0° et 359° sont cependant admis. Toute valeur inférieure ou supérieure peut compromettre l'exécution du sketch, si bien que le servo n'est plus commandé correctement. Quand vous donnez libre accès à une variable membre au moyen du modificateur `public`, aucune validation ne peut avoir lieu. Ce qui a été enregistré une fois produit immanquablement une réaction qui n'est pas forcément correcte. La solution du problème consiste à isoler les variables membres grâce à un modificateur d'accès `private` (privé). C'est le principe de l'encapsulation déjà évoqué qui est utilisé ici.



◀ Figure 8-5
Pas d'accès à une variable membre de la classe

C'est bien beau tout ça ! Mais comment fait-on pour accéder à la variable membre ? On y accède avec une méthode qui contient également un modificateur d'accès. Il doit cependant être `public` pour que l'accès fonctionne depuis l'extérieur. Le tout se présente comme sur la figure 8-6.

Figure 8-6 ▶
Accès à une variable membre de la classe par la méthode



On voit clairement que l'accès à la variable membre passe par la méthode, ceci étant un avantage et non pas un inconvénient. Vous pouvez maintenant procéder à la validation dans la méthode, seules des valeurs admises étant alors communiquées à la variable membre.

Mais comment la méthode a-t-elle accès à la variable membre `private` ? Le modificateur d'accès `private` signifie que l'accès depuis l'extérieur de la classe est impossible. Mais des membres de la classe comme les méthodes peuvent accéder à des membres déclarés `private`. Ils appartiennent tous à la classe et sont donc librement accessibles au sein de celle-ci. Pour faire court, les modificateurs d'accès gèrent l'accès aux membres de la classe.

Tableau 8-2 ▶
Modificateurs d'accès et leur signification

Modificateur d'accès	Description
public	L'accès aux variables membres et aux méthodes est possible depuis n'importe où dans le sketch. De tels membres constituent une interface publique de la classe.
private	L'accès aux variables membres et aux méthodes est réservé aux membres de la même classe.

Si vous voulez ajouter une classe à votre projet Arduino, mieux vaut créer un nouveau fichier se terminant par `.cpp` pour y stocker la définition de la classe. Vous verrez bientôt comment dans l'exemple concret de la bibliothèque-dé. Encore un peu de patience.

Une classe a besoin d'aide

Nous avons vu ce qu'une classe réalise et comment la créer en bonne et due forme. Mais je ne vous ai pas encore dit que la classe avait besoin d'un autre fichier très important. Celui-ci est appelé *fichier d'en-tête* et contient les déclarations (informations initiales ou préalables) pour la classe à concevoir. Si vous créez des variables membres ou des méthodes en C++, vous devez impérativement les faire connaître auprès du compilateur avant de les utiliser. C'est chose faite en définissant les variables et les

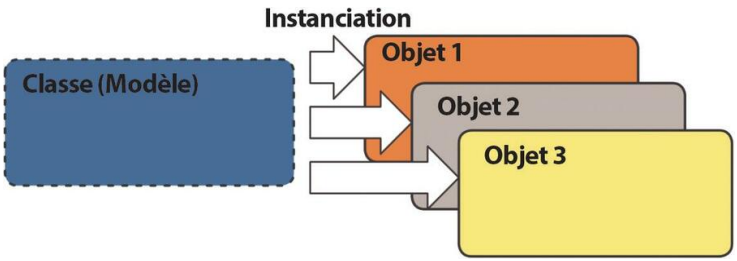
prototypes de fonction ou de méthode. Le fichier en question renferme également les consignes relatives aux modificateurs d'accès `public` et `private`. La construction formelle du fichier d'en-tête ressemble à celle de la définition de classe, à ceci près qu'il ne contient pas de formulation de code. Autrement dit, seules les signatures des méthodes sont mentionnées. Une signature se compose uniquement des informations initiales avec le nom de la méthode, le type d'objet renvoyé et la liste des paramètres. La construction générale est la suivante :

```
class <Nom> {
  public:
    // Membre public
  private:
    // Membre privé
};
```

Le nom de la classe (ici `<Nom>`) commence généralement par une majuscule, suivie de minuscules. La zone définissant le *membre public* vient après le mot-clé `public` suivi de deux-points. La zone définissant le *membre privé* vient après le mot-clé `private`, qui est suivi lui aussi de deux-points. Le fichier d'en-tête reçoit l'extension de nom `.h`.

Une classe devient un objet

Une fois créée par sa définition, une classe peut servir, comme lors de la déclaration d'une variable, de nouveau type de donnée. Ce procédé est appelé *instanciation*. Du point de vue du logiciel, la définition d'une classe ne veut pas dire qu'on a créé réellement un objet. Elle n'est qu'une sorte de modèle ou plan de construction qu'on peut utiliser pour concevoir un ou plusieurs objets.



◀ **Figure 8-7**
De la classe à l'objet

L'instanciation se fait de la manière suivante :

```
Nomclasse Nomobjet();
```

Nous avons dit que l'instanciation d'un objet avait tout de la déclaration de variable ordinaire. Mais vous noterez une paire de parenthèses derrière le nom que vous avez donné à l'objet. Est-ce une double faute de frappe ? Sûrement pas. Cette parenthèse double a naturellement son utilité. Une partie de ce projet va lui être consacrée, car elle est extrêmement importante pour l'instanciation.

Initialiser un objet : qu'est-ce qu'un constructeur ?

Une définition de classe contient en principe quelques variables membres qui serviront après l'instanciation. Pour qu'un objet puisse présenter un état initial bien défini, il s'avère judicieux de l'initialiser en temps voulu. Quel meilleur moment pour cette initialisation que directement lors de l'instanciation ? Aucun risque ainsi qu'elle soit oubliée et pose plus tard problème lors de l'exécution du sketch. Mais comment faire pour initialiser un objet ? Le mieux est d'employer une méthode qui prend cette tâche en charge.

Nous devons donc indiquer lors de l'instanciation une méthode à laquelle on donne certaines valeurs comme arguments. Mais comment savoir quelle méthode prendre ? Il faut appeler une méthode et lui donner le cas échéant quelques valeurs en passant. Mais quel nom lui donner ? La solution est à la fois très simple et géniale. La méthode pour initialiser un objet porte le même nom que la classe. Cette méthode étant très spéciale, elle porte aussi un nom à elle. On l'appelle *constructeur*. Comme son nom l'indique, elle construit en quelque sorte l'objet. Mais puisqu'il n'est pas impérativement nécessaire d'initialiser dès le début un objet avec certaines valeurs, elle n'a pas forcément de liste de paramètres. Elle se comporte alors comme une méthode à laquelle aucun paramètre n'est donné et qui n'a que la paire de parenthèses vide. Ceci répond à votre question concernant la paire de parenthèses que vous avez vue dans l'instanciation. Vous ne devez en aucun cas l'omettre ou l'oublier.

Il me faut maintenant être un peu plus concret pour vous en montrer la syntaxe. Voici le contenu du fichier d'en-tête de notre bibliothèque-dé :

```
class Dice{
      public:
         Dice(); // Constructeur
         //...
      private:
         // ...
};
```

Sous le modificateur d'accès `public` se trouve le constructeur qui porte le même nom que la classe. Il présente une paire de parenthèses vide, d'où son nom de *constructeur standard*.

Mais ne venons-nous pas de dire qu'on peut donner des arguments à un constructeur tout comme à une méthode pour initialiser l'objet ? La paire de parenthèses vide indique pourtant que le constructeur ne peut recevoir aucune valeur. Comment est-ce possible ? Et autre question concernant le type présumé d'objet retourné par une méthode : il n'a pas été indiqué pour le constructeur. Pourquoi ?

Effectivement, le constructeur ne peut accueillir aucune valeur sous cette forme. C'est une bonne introduction au prochain thème. Mais je vais d'abord répondre à votre question sur le type d'objet renvoyé manquant. Si une méthode renvoie une valeur à son appelant, le type de donnée en question doit naturellement être indiqué. Si aucun renvoi n'est prévu, le mot-clé `void` est utilisé. Revenons maintenant à notre constructeur. Il est appelé, non pas explicitement par une ligne d'instruction, mais implicitement par l'instanciation d'un objet. C'est pour cette raison que rien ne peut être retourné à un appelant et que le constructeur n'a pas même le type de renvoi `void`.

La surcharge

Ce que je vais vous dire là peut sembler déroutant à première vue : on peut définir un constructeur et bien entendu également des méthodes plusieurs fois avec le même nom.

Effectivement, c'est un peu difficile à croire, car c'est contraire au principe de clarté. Si par exemple une méthode apparaît deux fois avec le même nom dans un sketch, comment le compilateur peut-il savoir laquelle des deux est appelée ?

Mais il n'y a pas que le nom qui soit déterminant, il y a aussi la fameuse signature dont je vous ai parlé plus haut ! L'exemple suivant montre deux constructeurs acceptables qui portent le même nom, mais dont les signatures diffèrent :

```
Dice();
Dice(int, int, int, int);
```

Le premier constructeur représente le constructeur standard et sa paire de parenthèses vide, qui ne peut accueillir aucun argument. Le deuxième porte une toute autre signature, car il peut recevoir quatre valeurs du type `int`. Vous pouvez alors choisir entre deux variantes pour instancier un objet `Dice` :

```
Dice myDice();
```

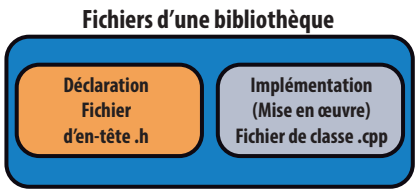
ou :

```
Dice myDice(8, 9, 10, 11);
```

Le compilateur est assez intelligent pour savoir quel constructeur il doit appeler.

La bibliothèque-dé

Toute cette introduction était nécessaire pour bien vous faire comprendre la création d'une bibliothèque Arduino. Ce deuxième projet de dé va servir de base pour constituer une bibliothèque. Il s'agit d'une variante améliorée avec commande des groupes de LED. Deux fichiers sont donc nécessaires pour réaliser la bibliothèque.



Le fichier d'en-tête

Commençons par le fichier d'en-tête, qui ne contient que les informations de prototype et ne présente aucune information de code explicite. Occupons-nous d'abord des membres de la classe qui sont nécessaires. Pour piloter les groupes de LED, il faut quatre broches numériques commandées par les variables membres:

- `pinGroupA` ;
- `pinGroupB` ;
- `pinGroupC` ;
- `pinGroupD`.

Ces informations seront transmises au moment de l'instanciation au constructeur, qui possède quatre paramètres du type `int`. Les variables membres sont déclarées privées (`private`), car elles ne sont traitées qu'en interne par une méthode appelée `roll`, qui n'a aucun argument et qui ne retourne rien. La classe reçoit le nom évocateur de `Dice` (dé).

```
#ifndef Dice_h
#define Dice_h
#include <Arduino.h>

class Dice {
      public:
         Dice(int, int, int, int); // Constructeur
         void roll(); // Méthode pour lancer le dé
      private:
          int GroupA; // Variable membre pour groupe de LED A
          int GroupB; // Variable membre pour groupe de LED B
          int GroupC; // Variable membre pour groupe de LED C
          int GroupD; // Variable membre pour groupe de LED D
};
#endif
```

Quelques informations supplémentaires méritant une explication ont été ajoutées à la définition de la classe. La classe tout entière a été enveloppée dans la structure suivante :

```
#ifndef Dice_h
#define Dice_h
     ...
#endif
```

Des inclusions multiples étant possibles quand il y a du code imbriqué, un moyen a été trouvé pour les empêcher et éviter une double compilation. Cette précaution a pour but de garantir une inclusion unique du fichier d'en-tête. Les instructions `#ifndef`, `#define` et `#endif` sont des *instructions de prétraitement*. `#ifndef`, qui introduit une compilation conditionnelle, est la forme abrégée de *if not defined* qui signifie « si non défini ». Si le terme `Dice_h` (nom du fichier d'en-tête avec un tiret bas) – appelé macro – n'a pas encore été défini, faites-le maintenant et exécutez les instructions dans le fichier d'en-tête. Si ce dernier était appelé une deuxième fois, la macro serait placée sous le nom et cette partie de la compilation serait rejetée.

Avez-vous compris pourquoi le constructeur n'indique que le type de donnée pour les paramètres et pourquoi le nom de la variable correspondante est absent ? Cela tient au fait que nous n'avons besoin ici que des informations de prototype. Le code en question apparaît plus tard avec une extension `.cpp` dans le fichier de classe.

Le fichier de classe

La véritable mise en œuvre du code est effectuée au moyen du fichier de classe présentant l'extension `.cpp` :

```
#include <Arduino.h>
#include "Dice.h"
#define WAITTIME 20

// Constructeur paramétré
Dice::Dice(int A, int B, int C, int D) {
GroupA = A;
GroupB = B;
GroupC = C;
GroupD = D;
pinMode(GroupA, OUTPUT);
pinMode(GroupB, OUTPUT);
pinMode(GroupC, OUTPUT);
pinMode(GroupD, OUTPUT);
}
// Méthode pour lancer le dé
void Dice::roll() {
int number = random(1, 7);
digitalWrite(GroupA, number%2!=0?HIGH:LOW);
digitalWrite(GroupB, number>1?HIGH:LOW);
digitalWrite(GroupC, number>3?HIGH:LOW);
digitalWrite(GroupD, number==6?HIGH:LOW);
delay(WAITTIME); // Ajouter une courte pause
}
```

Pour que la liaison vers le fichier d'en-tête précédemment créé soit possible, référence est faite à ce dernier au moyen de l'instruction `include` :

```
#include "Dice.h"
```

Son intégration intervient lors de la compilation. `include` est ici également nécessaire pour pouvoir utiliser ce qu'on appelle les éléments de langage Arduino. Passons maintenant au code, qui contient la mise en œuvre proprement dite. Commençons par le constructeur :

```
Dice::Dice(int A, int B, int C, int D) {
GroupA = A;
GroupB = B;
GroupC = C;
GroupD = D;
pinMode(GroupA, OUTPUT);
pinMode(GroupB, OUTPUT);
pinMode(GroupC, OUTPUT);
pinMode(GroupD, OUTPUT);
}
```

Vous remarquerez que la méthode `roll` est légèrement différente de celle du montage n° 10. Ici, aucune LED n'a été éteinte avant de commander l'allumage des nouvelles.

En effet, et ce n'est pas grave puisque les différents groupes de LED sont commandés par l'*opérateur conditionnel* `?` que vous rencontrerez dans le montage n° 9. Cet opérateur retourne soit `LOW` soit `HIGH` quand la condition a été évaluée, si bien que le groupe de LED correspondant a toujours le bon niveau et n'a pas besoin d'être remis auparavant sur `LOW`. Une autre chose susceptible de vous étonner est le préfixe `Dice::` qui précède aussi bien le nom de la structure que la méthode `roll`. Il s'agit du nom de la classe, qui permet au compilateur de savoir à quelle classe la définition de la méthode appartient. Cette dernière est qualifiée par cette notation. L'objet dé que nous voulons créer devant commander quatre groupes de LED, il est préférable de transmettre ces informations au moment de l'instanciation. Ce serait bien entendu possible aussi après la génération de l'objet, en utilisant une méthode distincte que nous appellerions par exemple `Init`. Mais le risque est grand que cette étape soit oubliée. C'est pourquoi le constructeur a été inventé. Voyons maintenant le sketch qui utilise cette bibliothèque.

Création des fichiers nécessaires

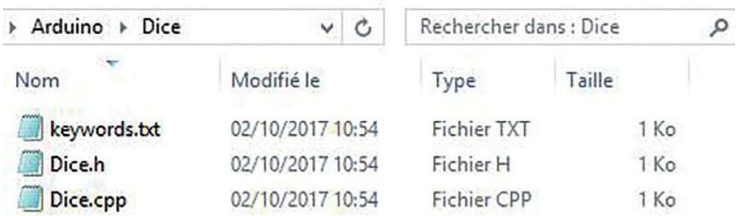
Je vous propose de programmer les deux fichiers de la bibliothèque `.h` et `.cpp` indépendamment de l'environnement de développement Arduino. Il existe pour ce faire de nombreux éditeurs, par exemple Notepad++ ou Programmers Notepad. Les deux fichiers sont stockés dans un répertoire au nom évocateur, par exemple `Dice`, que vous copierez ensuite dans le dossier des bibliothèques Arduino ensuite :

```
...\Arduino\libraries
```

Ce dossier contient les bibliothèques système. Vos bibliothèques personnelles seront placées dans le dossier :

```
C:\Users\<Username>\Documents\Arduino\libraries
```

L'environnement de développement Arduino est ensuite redémarré et la programmation du sketch peut commencer.



◀ **Figure 8-8**
Les fichiers des bibliothèques dans le système de fichiers

Mise en surbrillance de la syntaxe pour une nouvelle bibliothèque

Des types de données élémentaires (par exemple : `int`, `float` ou `char`) ou d'autres mots-clés (par exemple : `setup` ou `loop`) sont signalés en couleurs par l'environnement de développement. Il est possible, lors de la création de ses propres bibliothèques, de faire connaître à l'IDE des noms de classes ou de méthodes, pour qu'ils apparaissent alors aussi en couleurs. Un fichier nommé

```
keywords.txt
```

ayant obligatoirement une syntaxe spéciale doit être créé pour que cela fonctionne.

Commentaires

Des commentaires explicatifs sont introduits par le signe `#` (dièse) :

```
# Ceci est un commentaire
```

Types de données et classes (KEYWORD1)

Les types de données, mais aussi les noms de classes figurent en orange et doivent être définis en respectant la syntaxe suivante :

```
Nomclasse KEYWORD1
```

Méthodes et fonctions (KEYWORD2)

Méthodes et fonctions apparaissent en marron et doivent être définies en respectant la syntaxe suivante :

```
Méthode KEYWORD2
```

Constantes (LITERAL1)

Les constantes sont en bleu et sont créées comme suit :

```
Constante LITERAL1
```

Voici maintenant le contenu du fichier `keywords.txt` de notre bibliothèque-dé :

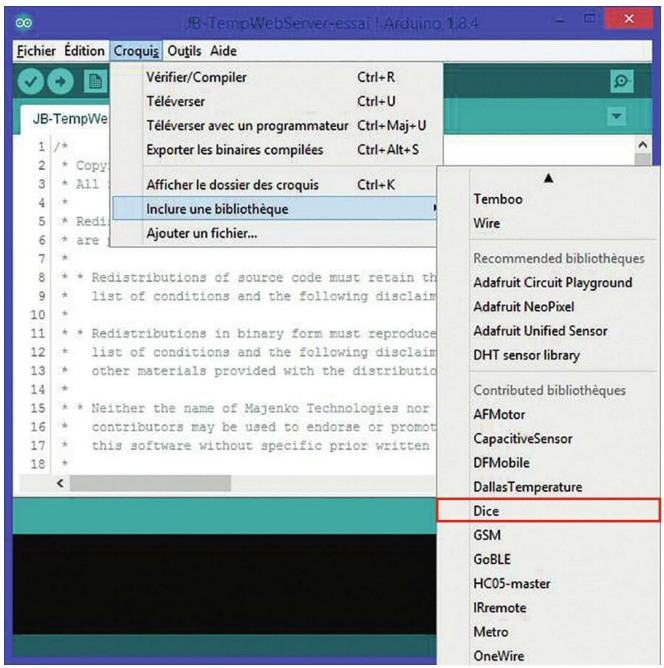
```
#--------------------------------------
# Affectation des couleurs pour la bibliothèque Dice
#--------------------------------------
#--------------------------------------
# KEYWORD1 pour types de données ou classes
#--------------------------------------
Dice  KEYWORD1
#--------------------------------------
# KEYWORD2 pour méthodes et fonctions
#--------------------------------------
roll KEYWORD2
#--------------------------------------
# LITERAL1 pour constantes
#--------------------------------------
```

Utilisation de la bibliothèque

Le fait que la bibliothèque-dé se trouve dans le répertoire

```
C:\Users\<Username>\Documents\Arduino\libraries
```

vous permet de la retrouver dans la dernière entrée du menu.



◀ Figure 8-9
Importer la bibliothèque-dé

Le terme *importer* est quelque peu mal venu puisque absolument rien n'est importé à ce moment précis. Seule la ligne suivante est ajoutée dans votre fenêtre de sketch :

```
#include <Dice.h>
```

La ligne `include` est impérative pour pouvoir accéder à la fonctionnalité de la bibliothèque-dé. Comment le compilateur saurait-il sinon à quelle bibliothèque il doit accéder ? Les différentes bibliothèques disponibles ne sont pas incluses par l'opération du Saint-Esprit ! Passons maintenant à l'instanciation, qui élève la définition de classe au statut d'objet réel. L'objet créé `myDice` est également appelé *variable d'instance*. Ce terme revient souvent dans la littérature.

```
Dice myDice(8, 9, 10, 11);
```

Les valeurs transmises `8`, `9`, `10` et `11` figurent les broches numériques auxquelles les groupes de LED sont reliés. Un objet dé a ainsi été initialisé de manière à pouvoir opérer en interne quand la méthode pour lancer le dé est appelée. Les arguments sont transmis dans l'ordre indiqué.

```
Dice myDice(8, 9, 10, 11);

Dice::Dice(int A, int B, int C, int D) {
{
   ...
}
```

Ils sont stockés dans les variables locales `A`, `B`, `C` et `D`, qui sont à leur tour transmises aux variables membres `GroupA`, `GroupB`, `GroupC` et `GroupD`. Vient ensuite l'appel de la méthode à condition qu'un potentiel `HIGH` soit appliqué à l'entrée numérique, appel qui peut se faire par le bouton-poussoir raccordé.

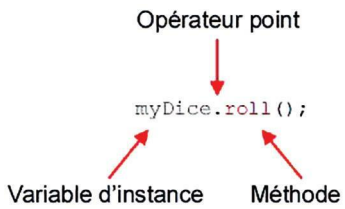
```
void setup() {
pinMode(13, INPUT); // Pas obligatoire – oui, mais pourquoi !?
}

void loop() {
   if(digitalRead(13) == HIGH)
      myDice.roll();
}
```

On voit ici aussi que la mise en surbrillance de la syntaxe fonctionne puisque le nom de classe et la méthode sont en couleurs. La méthode `roll` étant

un membre de la définition de classe `myDice`, une liaison vers la classe doit être établie en cas d'appel de celle-ci. Un appel par

```
roll();
```

provoquerait ici une erreur. La relation est établie par l'*opérateur point* inséré entre classe et méthode et faisant office de lien.



Importation de bibliothèques

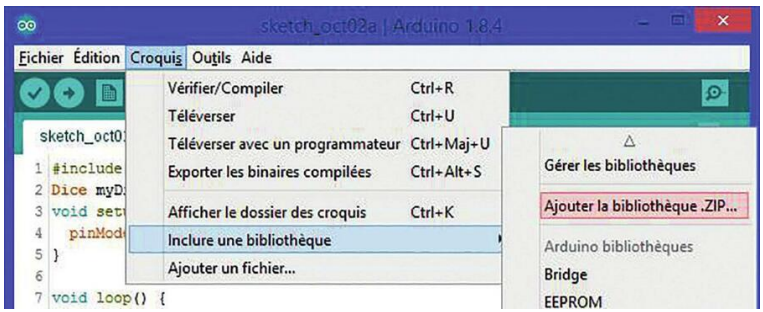
Il est aussi possible de gérer les bibliothèques autrement. Nous avons créé un dossier nommé `Dice` pour la bibliothèque-dé à l'intérieur duquel nous avons enregistré les fichiers :

- `dice.h`
- `dice.h`
- `keywords.txt`

Puis nous avons copié ce dossier dans le répertoire

```
C:\Users\<Username>\Documents\Arduino\libraries
```

afin qu'il soit accessible à l'aide de la commande *Sketch | Import Library* après le redémarrage de l'IDE Arduino. Vous avez certainement remarqué qu'au sommet du menu figure une commande qui permet d'ajouter d'autres bibliothèques :



◀ Figure 8-10
Ajouter une bibliothèque

En sélectionnant cette commande, vous avez la possibilité d'intégrer une bibliothèque compressée, que vous aurez préalablement téléchargée sur Internet, par exemple, dans l'environnement Arduino ou dans le système de fichiers. Il n'est pas nécessaire de décompresser préalablement le fichier, car l'IDE s'en charge automatiquement.

Vous trouverez plus d'informations sur l'importation de bibliothèques à l'adresse suivante :

https://www.arduino.cc/en/Guide/Libraries

Qu'avez-vous appris ?

- Je reconnais que les choses sont devenues un peu plus difficiles dans ce projet, mais le jeu en vaut la chandelle. Vous en savez maintenant plus sur le paradigme de programmation orientée objet.
- Vous savez bien faire la différence entre une classe et un objet.
- Dans la POO, *méthode* remplace *fonction* et *variable membre* remplace *variable*.
- Le constructeur est une méthode avec une tâche particulière. Il initialise l'objet de manière à obtenir un état de départ bien défini.
- Les différents modificateurs d'accès `public` ou `private` réglementent l'accès aux membres de l'objet, `private` assurant l'encapsulation des membres.
- Vous connaissez les informations de code qu'un fichier d'en-tête ou `.cpp` doit contenir.
- L'objet instancié à partir d'une classe est également appelé variable d'instance.
- Un opérateur point, ajouté après le nom de la variable d'instance et servant quasiment de lien entre les deux, est utilisé pour accéder aux variables membres et aux méthodes.
- Vous savez comment créer une bibliothèque Arduino et à quel endroit copier celle-ci dans le système de fichiers pour pouvoir y accéder à tout moment.
- Vous avez enfin appris à configurer certaines méthodes en tant que mots-clés avec un marquage couleur. L'enregistrement s'effectue à un emplacement auquel se trouvent habituellement aussi les sketches, mais à l'intérieur du dossier `libraries`.
- Enfin, vous avez vu comment configurer certaines méthodes en tant que mots-clés en couleur.

Les feux de circulation

Dans le premier montage, nous avions vu comment faire clignoter une seule LED. Maintenant, nous allons en allumer trois. Elles ne brilleront pas toutes d'un coup, bien sûr ! En fait, nous allons construire des feux de circulation. La commande s'effectue par phases que nous étudierons en détail. Ensuite, nous compléterons nos feux de circulation par des feux pour piétons. Voyons d'abord la liste des composants.

Composants nécessaires

Ce montage nécessite les composants suivants.

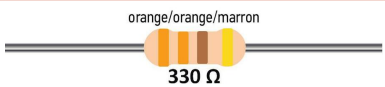
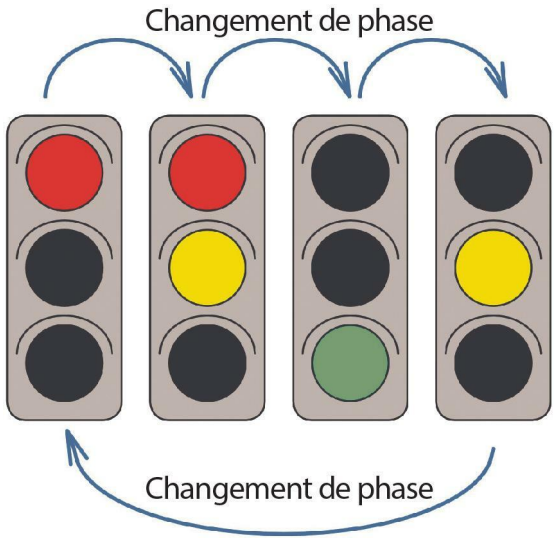
Composant	
2 LED rouges	
1 LED orange	
2 LED vertes	
5 résistances de 330 Ω	
1 bouton-poussoir miniature	
1 résistance de 10 kΩ	

Tableau 9-1
Liste des composants

Phases de signalisation

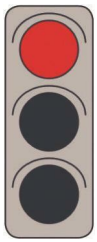
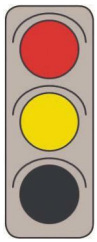
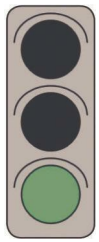
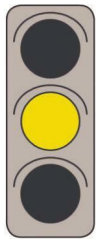
Voyons d'abord les différentes phases de signalisation possibles :

Figure 9-1 ▶
Les différentes phases de signalisation



Les différentes phases sont parcourues de gauche à droite, le cycle reprenant ensuite au début. Chacune phase a une durée d'affichage définie pouvant être ajustée individuellement. Voici un exemple de durées d'allumage.

Tableau 9-2 ▶
Phases avec durées d'allumage

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

Schéma

Examinons le schéma de la page suivante.

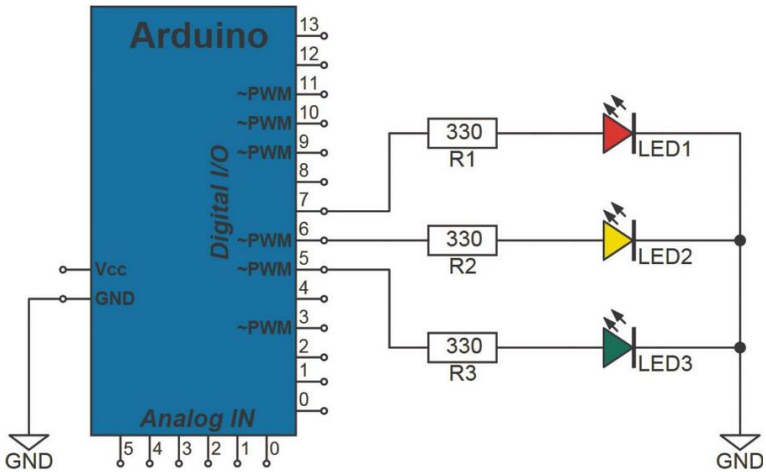


Figure 9-2
Schéma

Chaque LED est commandée par une résistance série de 330 Ω. Le calcul d'une résistance série pour une LED ne vous pose plus de problème.

Réalisation du circuit

L'illustration suivante présente la réalisation du circuit sur une plaque d'essais :

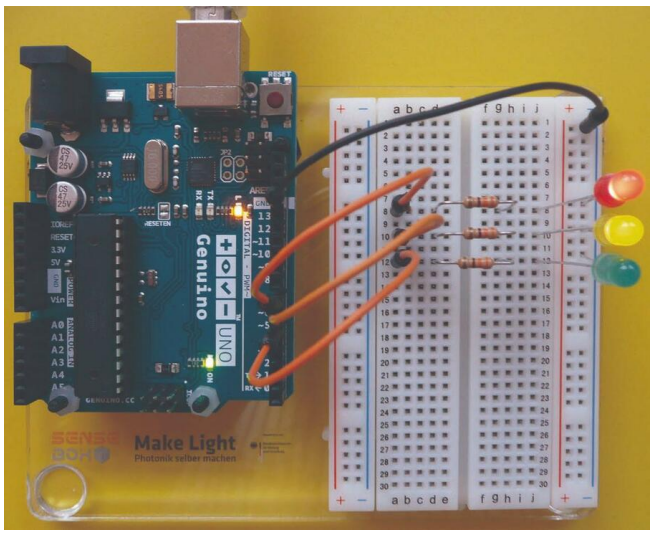


Figure 9-3
Réalisation du circuit

Les sorties numériques 5, 6 et 7 sont raccordées à l'aide de cavaliers flexibles aux résistances série afin de commander les LED.

Sketch Arduino

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

```
const int ledPinRed   = 7;   // Broche 7 commande la LED rouge
const int ledPinOrange  = 6; // Broche 6 commande la LED orange
const int ledPinGreen = 5;   // Broche 5 commande la LED verte

#define DELAY1 10000 // Pause 1, 10 secondes
#define DELAY2 2000  // Pause 2, 2 secondes
#define DELAY3 3000  // Pause 3, 3 secondes

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
}
```

Peut-être avez-vous remarqué quelque chose d'inhabituel dans la définition des broches au début du sketch. J'ai déjà présenté les avantages de ce mode d'écriture lors du premier montage. Pourtant, il semble y avoir ici trois formes d'écritures différentes. Voyons cela de plus près. Supposons que nous voulions définir la broche 13 au début du sketch. Nous pourrions employer les formulations suivantes :

```
// Définition d'une variable (occupe de la RAM)
int ledPin = 13;
// Définition d'une constante (n'occupe pas de RAM)
const int ledPin = 13;
// Directive de prétraitement (n'occupe pas de RAM)
#define LEDPIN 13
```

Les différences résident dans les détails. Dans la première variante, une variable du type de donnée Integer est définie, ce qui occupe de la place

dans la mémoire vive. Quand les sketches sont assez courts, cela ne pose pas trop de problèmes. D'ailleurs, de nombreux exemples utilisent cette possibilité. La seconde variante utilise aussi le mot-clé `const`, ce qui signifie qu'une constante est définie. Contrairement à la méthode précédente, la valeur ne peut alors pas être modifiée. Il s'agit presque d'une variable de type *lecture seule* qui n'occupe pas d'espace dans la mémoire vive. C'est donc un bon choix pour économiser de la capacité de stockage. Dans le dernier cas, il s'agit d'une directive de prétraitement qui ne doit pas se terminer par un point-virgule. Cette directive remplace chaque occurrence de l'indicateur défini `LEDPIN` à l'intérieur du sketch par la séquence de caractères suivante `13` et elle n'utilise pas non plus de mémoire. Nous pouvons donc en conclure que la deuxième méthode employant le mot-clé `const` est le meilleur choix.

Vous trouverez de plus amples informations aux adresses suivantes :

https://www.arduino.cc/en/Reference/int

https://www.arduino.cc/en/Reference/Const

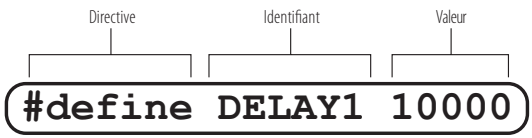
https://www.arduino.cc/en/pmwiki.php?n=Reference/Define

Revue de code

Nous utilisons des constantes numériques ou symboliques pour pouvoir modifier facilement certaines valeurs du sketch. Les modifications sont effectuées de façon centralisée, ce qui présente l'avantage qu'elles sont appliquées automatiquement au reste du code. Cette méthode réduit le travail de saisie, tout en rendant le code plus lisible et en réduisant le risque d'erreurs qui pourraient se glisser dans le code.

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 celle du haut de la page suivante.

Figure 9-4 ▶
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 !*

Mais pourquoi n'avons-nous 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

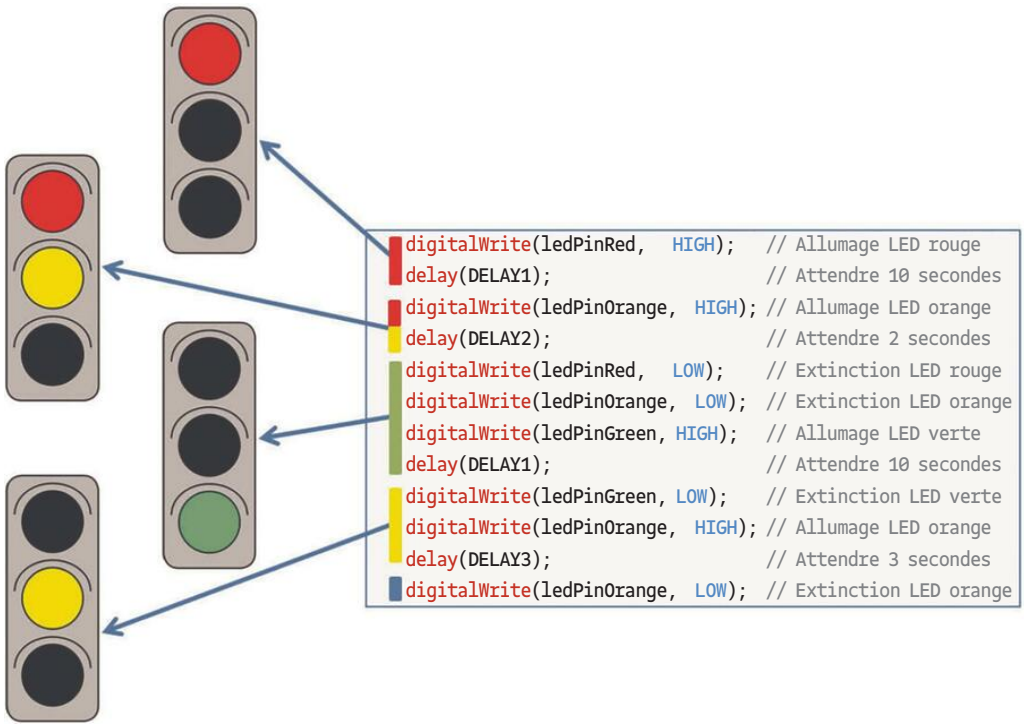
```
int ledPinRed = 7;
```

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 variables quand il s'agit de broches, et la directive `#define` pour les constantes. Revenons à notre sketch et voyons comment il fonctionne.

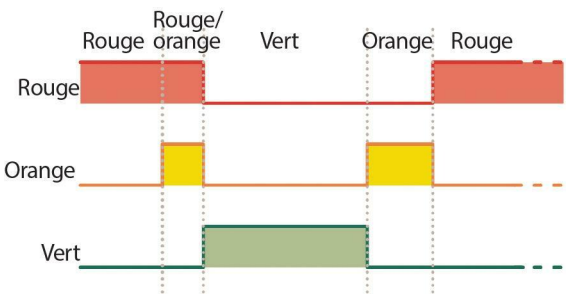
Pour que les feux de signalisation fonctionnent correctement, il faut non seulement penser à allumer les diverses LED, mais aussi à les éteindre. Le traitement s'effectue en continu, de haut en bas, à l'intérieur de la fonction `loop`. Examinons les différentes phases avec leurs commandes dans le sketch.



```
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
```

▲ **Figure 9-5**
Commande des différentes phases de signalisation

Au passage de la phase 1 à la phase 2, seule une LED orange vient s'ajouter à la LED rouge. La rouge continue à briller. 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. Ensuite, au passage de la phase 4 à la phase 1, lorsque les phases recommencent au début, la LED orange doit s'éteindre. Jetez un coup d'œil au chronogramme pour voir comment les LED sont allumées à tour de rôle pendant les différentes phases.

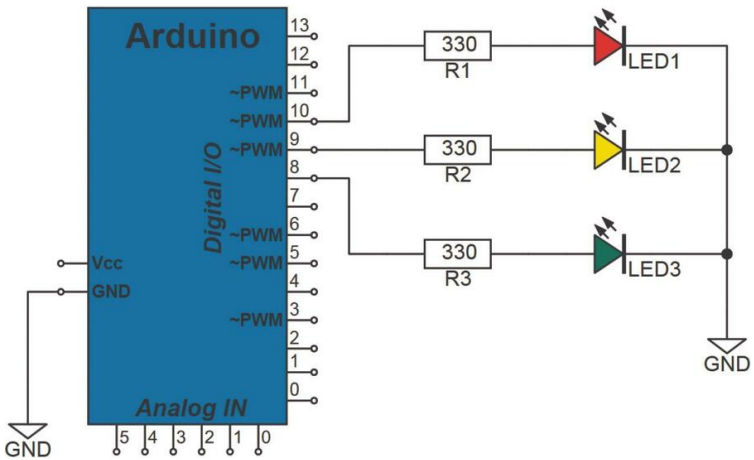


◀ **Figure 9-6**
Chronogramme des feux de circulation

Variante utilisant le port B

Examinons une variante intéressante qui met en pratique les connaissances que nous avons acquises sur la commande directe du port à l'aide du registre afin de manipuler directement les différentes broches. Nous utilisons de nouveau le port B et nous modifions le circuit comme illustré :

Figure 9-7 ▶
Schéma



Le code du sketch de commande est le suivant :

```
#define DELAY1 10000 // Pause 1, 10 secondes
#define DELAY2 2000  // Pause 2, 2 secondes
#define DELAY3 3000  // Pause 3, 3 secondes

void setup() {
  DDRB  = 0b00000111; // Broches 8, 9, 10 comme sortie
}

void loop() {
  PORTB = 0b00000100;  // Mise de la broche 10 au niveau HIGH (rouge)
  delay(DELAY1);       // Attendre 10 secondes
  PORTB = 0b00000110;  // Mise des 10,9 au niveau HIGH (rouge, orange)
  delay(DELAY2);       // Attendre 2 secondes
  PORTB = 0b00000001;  // Mise de la broche 8 au niveau HIGH (vert)
  delay(DELAY1);       // Attendre 10 secondes
  PORTB = 0b00000010;  // Mise de la broche 9 au niveau HIGH (orange)
  delay(DELAY3);       // Attendre 3 secondes
}
```

Les trois bits inférieurs sont chargés de commander les trois LED. L'avantage de ce mode de programmation est évident : une seule commande `PORTB` permet de modifier le niveau de plusieurs broches. Il n'est donc plus nécessaire de s'assurer que les autres LED sont éteintes avant d'en allumer une nouvelle.

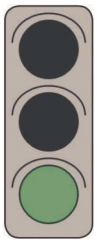
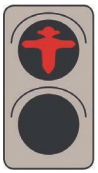
Des feux de circulation interactifs

Jusque-là, le sketch était relativement simple. Nous allons donc le modifier légèrement. 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.

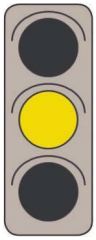
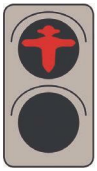
La situation de départ ressemble à ceci :

1re phase

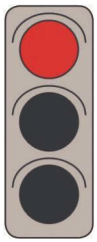
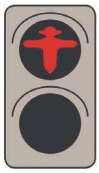
Automobiliste	Piéton	Explications
		Ces deux signaux lumineux restent allumés jusqu'à ce qu'un piéton s'approche et appuie sur le bouton gérant les feux. C'est alors seulement que les changements de phase se déclenchent et font en sorte que le feu passe au rouge pour les automobilistes et au vert pour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

Examinons cela de plus près.

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 piéton. Durée : 1 s

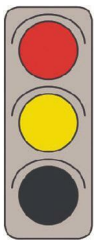
4e phase

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

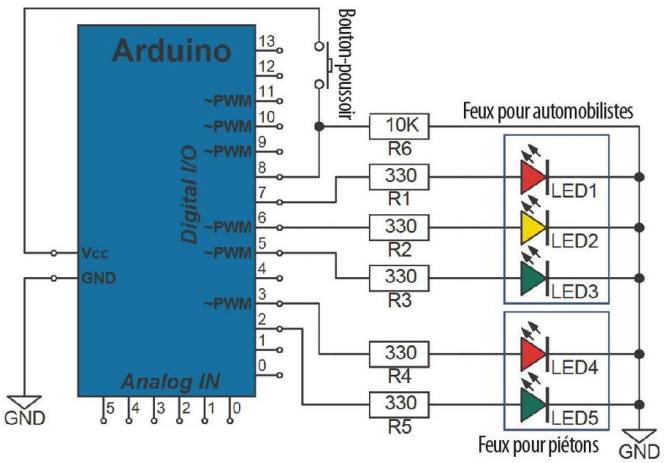
Automobiliste	Piéton	Explications
		Le feu passe du rouge à l'orange pour les automobilistes, ce qui les avertit que le feu va bientôt passer au vert. Durée : 2 s

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

Schéma

Le schéma est un peu plus complexe.

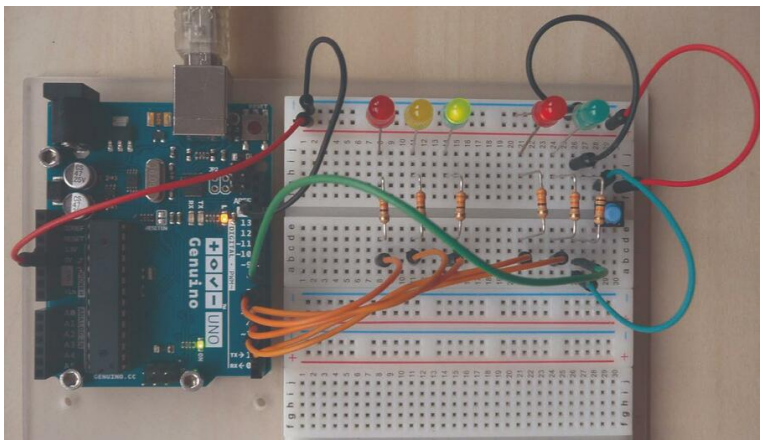


◀ Figure 9-8
Circuit des feux de circulation interactifs

Réalisation du circuit

Sur la plaque d'essais sont venues s'ajouter les deux LED supplémentaires avec les résistances série, ainsi que le bouton-poussoir avec la résistance pull-up :

Figure 9-9 ▶
Réalisation du circuit



Le sketch élargi est 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 ledPinRedCar = 7;    // La broche 7 commande la LED rouge
                         // (feux pour les automobilistes)
int ledPinOrangeCar = 6; // La broche 6 commande la LED orange
int ledPinGreenCar = 5;  // La broche 5 commande la LED verte
int ledPinRedWalk = 3;   // La broche 3 commande la LED rouge
                         // (feux pour les piétons)
int ledPinGreenWalk = 2;  // La broche 2 commande la LED verte
int buttonPinLight = 8;  // Bouton gérant les deux reliés à la broche 8
int buttonLightValue = LOW; // Variable pour l'état du bouton
                            // gérant les feux

void lightChange() {
   digitalWrite(ledPinGreenCar, LOW);
   digitalWrite(ledPinOrangeCar, HIGH); delay(DELAY3);
   digitalWrite(ledPinOrangeCar, LOW);
   digitalWrite(ledPinRedCar, HIGH); delay(DELAY1);
   digitalWrite(ledPinRedWalk, LOW);
   digitalWrite(ledPinGreenWalk, HIGH); delay(DELAY0);
   digitalWrite(ledPinGreenWalk, LOW);
   digitalWrite(ledPinRedWalk, HIGH); delay(DELAY1);
   digitalWrite(ledPinOrangeCar, HIGH); delay(DELAY2);
   digitalWrite(ledPinRedCar, LOW);
```

```
  digitalWrite(ledPinOrangeCar, LOW);
  digitalWrite(ledPinGreenCar, HIGH);
}

void setup() {
  pinMode(ledPinRedCar, OUTPUT);      // Broche comme sortie
  pinMode(ledPinOrangeCar, OUTPUT);   // Broche comme sortie
  pinMode(ledPinGreenCar, OUTPUT);    // Broche comme sortie
  pinMode(ledPinRedWalk, OUTPUT);     // Broche comme sortie
  pinMode(ledPinGreenWalk, OUTPUT);   // Broche comme sortie
  pinMode(buttonPinLight, INPUT);     // Broche comme entrée
  digitalWrite(ledPinGreenCar, 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 pressé, fonction appelée
  if(buttonLightValue == HIGH)
    lightChange();
}
```

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. 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(ledPinGreenCar, 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` :

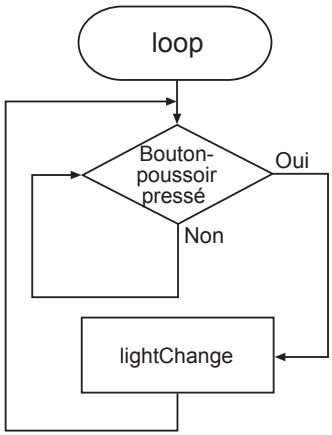
```
if(buttonLightValue == HIGH)
  lightChange();
```

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 nous appuyons 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 la question.

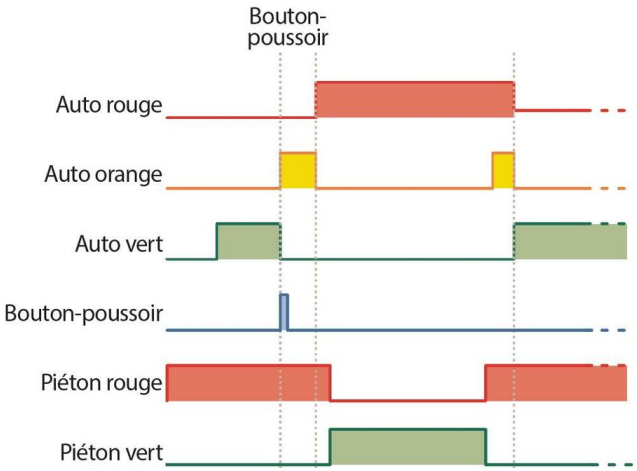
Figure 9-10 ▶
Appel de la fonction `lightChange`



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 lancés 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 9-11 ▶
Appel et retour

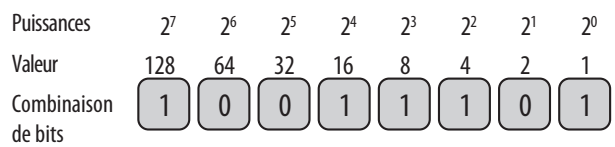
Avant de passer au schéma, voici encore un chronogramme montrant les différentes durées d'allumage l'une par rapport à l'autre. La situation de départ nous montre que le feu est vert pour les automobilistes et rouge pour les piétons. Un piéton ayant l'intention de traverser la route à un endroit supposé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



◀ **Figure 9-12**
Chronogramme des feux interactifs

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. L'ordinateur et la 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° 7 sur l'extension de port numérique. En voici les grandes lignes.



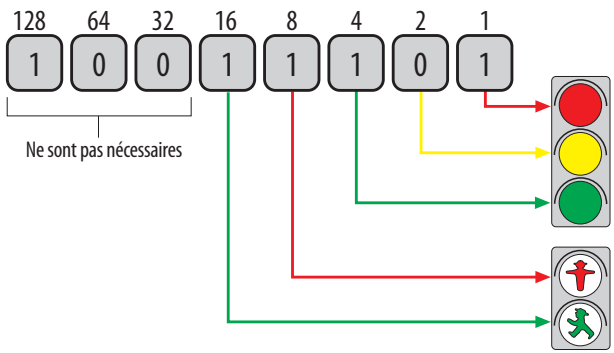
◀ **Figure 9-13**
Combinaison binaire pour le nombre entier 157

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

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

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. On pourrait faire encore plus clair :

Figure 9-14 ▶
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 9-3 les nombres décimaux correspondants, que j'ai déterminés à partir des différentes phases.

Tableau 9-3 ▶
Nombres décimaux pour commander les LED

	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
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 suivant montre que le résultat n'est `1` que si *les deux* opérandes ont `1` pour valeur.

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

◀ Tableau 9-4
Opération ET bit à bit

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.

	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

◀ Tableau 9-5
Contrôle de correspondance du bit

Le deuxième opérande avec la valeur décimale `8` sert en quelque sorte de filtre. Il vérifie, seulement dans la position du bit de poids 23, qu'un `1` se trouve bien dans le premier opérande. C'est le cas dans notre exemple et le résultat est `8`. Le tableau suivant 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 9-6
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.

```
Condition?Instruction1:Instruction2
```

◀ Figure 9-15
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(ledPinRedCar,   (lightValue&1) == 1?HIGH:LOW);
digitalWrite(ledPinOrangeCar,  (lightValue&2) == 2?HIGH:LOW);
digitalWrite(ledPinGreenCar, (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.

Vous vous demandez peut-être comment on obtient les différentes durées d'allumage des diverses phases de signalisation, car on ne voit pas l'instruction `delay` qui sert à définir les pauses.

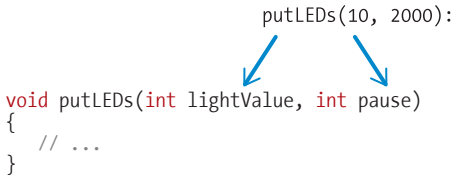
Effectivement, aussi ces lignes de code sont-elles insérées dans une fonction à part et complétées par `lightValue` et une deuxième valeur pour la fonction `delay`. Cela donne :

```
void putLEDs(int lightValue, int pause) {
  digitalWrite(ledPinRedCar,   (lightValue&1) == 1?HIGH:LOW);
  digitalWrite(ledPinOrangeCar,  (lightValue&2) == 2?HIGH:LOW);
  digitalWrite(ledPinGreenCar, (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 9-3. 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 À L'ORDRE DES ARGUMENTS

Respectez impérativement l'ordre des arguments lors de l'appel de la fonction. Cela ne ferait pas planter le sketch, mais le circuit ne réagirait pas comme prévu. Il faut donc que :

- le nombre des arguments coïncide avec celui des paramètres ;
- les *types de données* des arguments transmis correspondent à ceux des paramètres ;
- l'ordre soit respecté lors de l'appel.

Nous avons employé encore une fois l'expression « variable locale ». Mais avez-vous bien saisi la différence entre variable *locale* et variable *globale* ?

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 ledPinRedCar = 7;    // Broche 7 commande la LED rouge (feux pour
                         // les automobilistes)
// ...
```

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

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

Les *variables locales* sont déclarées ou initialisées dans des fonctions ou, par exemple, dans une boucle `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 quit-

tée, 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.

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.

Dans le répertoire suivant :

```
...\Arduino\hardware\arduino\avr\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 :

```
36  #define HIGH 0x1
37  #define LOW  0x0
38
39  #define INPUT 0x0
40  #define OUTPUT 0x1
41
42  #define true 0x1
43  #define false 0x0
44
45  #define PI 3.14159265358979323846264338327950
46  #define HALF_PI 1.57079632679489661923132169163975
47  #define TWO_PI 6.28318530717958647692528676655900
48  #define DEG_TO_RAD 0.017453292519943295769236907684886
49  #define RAD_TO_DEG 57.295779513082320876798154814105
50
51  #define SERIAL  0x0
52  #define DISPLAY 0x1
53
54  #define LSBFIRST 0
55  #define MSBFIRST 1
```

Cela ne vous rappelle rien ? Vous en saurez bientôt davantage sur ce qu'est un fichier d'en-tête. 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 ?
- Pas de 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 sortie numérique à l'aide de la fonction `digitalRead`.
- Vous avez réalisé un circuit pour feux de circulation à la fois simple, car il lance automatiquement les différents changements de phase indépendamment des influences externes, mais aussi interactif car il réagit 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.

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.

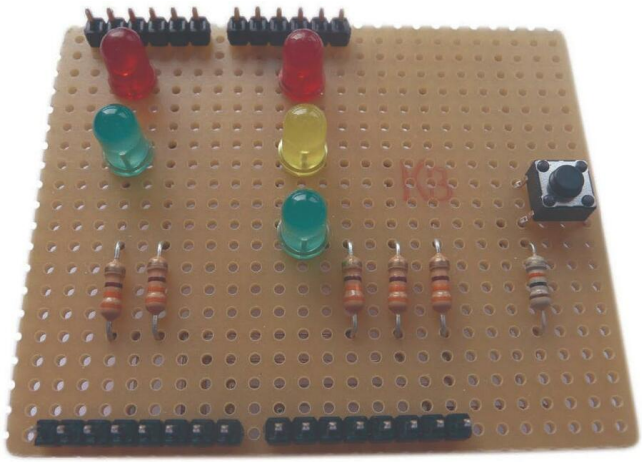


Figure 9-16 ▶
Circuit pour feux de circulation avec feux pour piétons

Montage 10

Le dé électronique

En microélectronique, il existe de multiples applications de jeux de dés ou de plateaux connus dont l'adaptation ne manquera pas de vous divertir. La construction d'un circuit et la programmation d'un dé sont de beaux projets qui nous permettront d'aborder différentes problématiques. Et même si cela paraît assez simple, vous devrez néanmoins éviter de tomber dans quelques pièges qui seront autant d'occasions d'apprentissage.

Vous apprendrez notamment à utiliser des tableaux bidimensionnels. Vous utiliserez le moniteur série non seulement pour saisir le contenu de variables, mais aussi pour rechercher des erreurs dans le code. Enfin, je vous expliquerai comment calculer la résistance lorsque deux LED sont branchées l'une derrière l'autre.

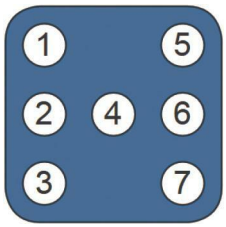
Qu'est-ce qu'un dé électronique ?

Bien que les derniers montages vous aient donné quelques bases pour programmer la carte Arduino, vous pensez sûrement être encore loin du compte... Aussi allons-nous appliquer, approfondir et élargir nos connaissances en étudiant quelques circuits intéressants. La construction d'un dé électronique est toujours plaisante. Il y a quelques années de cela, quand les microprocesseurs n'existaient pas ou étaient hors de prix, on utilisait plusieurs circuits intégrés. Vous trouverez pour ce faire de nombreuses instructions de bricolage sur Internet.

Notre but est ici de commander le dé électronique avec la seule carte Arduino. Tout le monde a déjà joué aux dés, que ce soit au 421, au 5 000 ou au Yams. Aussi notre prochain circuit sera celui d'un dé électronique.

Il se compose d'une unité d'affichage avec sept LED et un bouton-poussoir pour lancer le dé. Voici d'abord la disposition des LED qui est celle des points d'un véritable dé. Les points portent tous un numéro pour mieux se repérer au moment de commander les LED. Le numéro 1 se trouve en haut à gauche, la numérotation se poursuivant vers le bas puis vers la droite jusqu'au numéro 7 qui se trouve en bas à droite.

Figure 10-1 ▶
Numérotation des points du dé



Même si un dé ne comporte habituellement que six faces, nous avons néanmoins besoin de sept LED pour pouvoir représenter les six résultats pouvant être produits par un dé. Bien qu'il soit possible de représenter le quatre avec les quatre LED des coins, par exemple, nous avons aussi besoin de la LED au milieu du circuit pour le un, le trois et le cinq.

Notre construction doit comporter un bouton-poussoir qui lance le dé. Lorsqu'on appuie dessus, toutes les LED clignotent irrégulièrement et quand on le relâche, l'affichage s'arrête sur une certaine combinaison de LED, laquelle représente le nombre obtenu. Les différentes combinaisons de points sont répertoriées dans le tableau ci-dessous.

Tableau 10-1 ▶
Quelles LED s'allument pour quel nombre ?

Dé	Nombre	LED						
		1	2	3	4	5	6	7
	1				✓			
	2	✓						✓
	3	✓			✓			✓
	4	✓		✓		✓		✓
	5	✓		✓	✓	✓		✓
	6	✓	✓	✓		✓	✓	✓

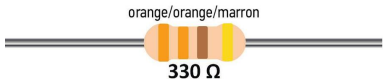
Il est certes tout à fait possible de construire le circuit sur votre plaque d'essais, mais ce n'est pas toujours simple compte tenu de la symétrie des LED. Dans un second montage, nous construirons le circuit sur une carte spéciale appelée *shield*, que nous insérerons au-dessus de la carte Arduino. C'est la manière la plus propre et la plus pratique de fabriquer un dé électronique durable. Mais utilisons d'abord la plaque d'essais. Quel matériel nous faut-il ?

PROTOTYPAGE

Nous réalisons tout d'abord le circuit sur la plaque d'essais en enfichant les LED, les résistances et le bouton-poussoir, ainsi que les câbles, uniquement sur la plaque. Cela nous permet de vérifier que le circuit fonctionne. Si tout va bien, nous pouvons ensuite souder le circuit sur une platine pour lui donner une forme durable. Le fait de tester un circuit sur une plaque d'essais se nomme le *prototypage*.

Composants nécessaires

Ce montage nécessite les composants suivants.

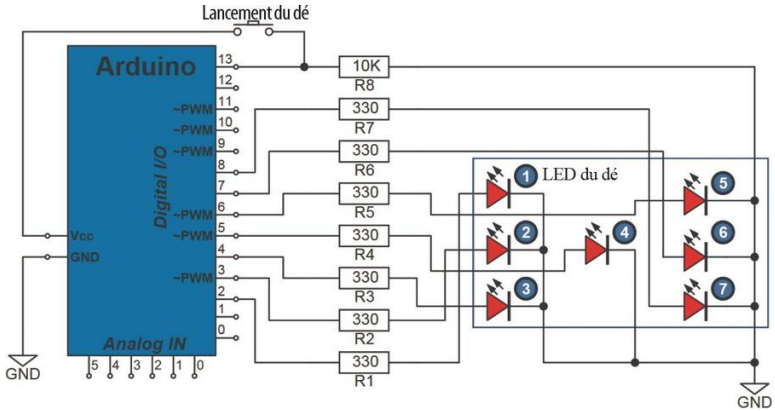
Composant	
7 LED rouges	
7 résistances de 330 Ω	
1 résistance de 10 kΩ	
1 bouton-poussoir miniature	

◀ Tableau 10-2
Liste des composants

Schéma

Le schéma montre les 7 LED du dé avec leur résistance série de 330 Ω et le bouton-poussoir avec sa résistance pull-down.

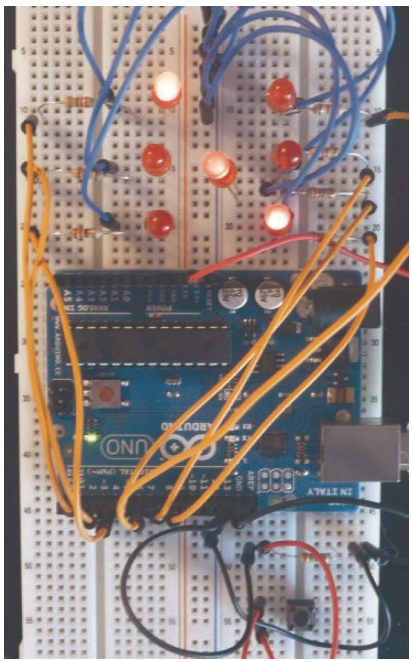
Figure 10-2 ▶
Carte Arduino commandant chacune des LED du dé



Réalisation du circuit

La figure suivante montre la construction du circuit sur une seule plaque, mais il a fallu « jongler » un peu pour y arriver. Ceci dit, il devrait très bien fonctionner.

Figure 10-3 ▶
Réalisation du dé électronique sur une plaque d'essais



Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Code du sketch

Voici le code du sketch pour commander le dé électronique :

```
#define WAITTIME 20
int pips[6][7] = {{0, 0, 0, 1, 0, 0, 0}, // Nombre sorti 1
                  {1, 0, 0, 0, 0, 0, 1},  // Nombre sorti 2
                  {1, 0, 0, 1, 0, 0, 1},  // Nombre sorti 3
                  {1, 0, 1, 0, 1, 0, 1},  // Nombre sorti 4
                  {1, 0, 1, 1, 1, 0, 1},  // Nombre sorti 5
                  {1, 1, 1, 0, 1, 1, 1}}; // Nombre sorti 6
int pin[] = {2, 3, 4, 5, 6, 7, 8};
int pinOffset = 2;  // Première LED sur broche 2
int buttonPin = 13; // Bouton-poussoir sur broche 13

void displayPips(int value) {
  for(int i = 0; i < 7; i++)
    digitalWrite(i + pinOffset, (pips[value - 1][i] == 1)?HIGH:LOW);
  delay(WAITTIME); // Ajouter une courte pause
}

void setup() {
  for(int i = 0; i < 7; i++)
    pinMode(pin[i], OUTPUT);
  pinMode(buttonPin, INPUT);
}

void loop() {
  if(digitalRead(buttonPin) == HIGH)
    displayPips(random(1, 7)); // Générer un nombre aléatoire entre 1 et 6
}
```

Examinons la signification de ce sketch.

Revue de code

La programmation est déjà plus compliquée et nous n'avons pas seulement affaire cette fois à un tableau unidimensionnel comme dans le montage n° 6 sur le séquenceur de lumière. Un tableau bidimensionnel est ici nécessaire pour mémoriser les numéros des LED qui doivent s'allumer en fonction du nombre obtenu. Rappelons-nous encore une fois, par l'intermédiaire de la figure suivante, comment un tableau unidimensionnel fonctionne et comment nous pouvons y accéder.

◀ Figure 10-4
Tableau unidimensionnel

La déclaration et l'initialisation du tableau sont assurées par la ligne suivante :

```
int ledPin[] = {7, 8, 9, 10, 11, 12, 13};
```

Le tableau contient ici sept éléments. Un tableau unidimensionnel est reconnaissable à sa paire de crochets derrière le nom de variable. On accède à un élément particulier en indiquant l'index entre les crochets. Vous écrivez donc ce qui suit pour accéder au 4e élément :

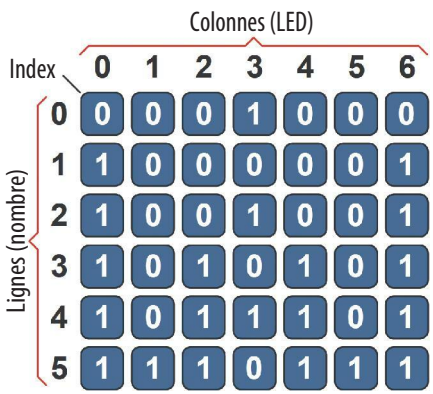
```
ledPin[3]
```

N'oubliez pas que l'on compte à partir de 0 !

Comme la première LED de notre montage ne se trouve pas sur la broche 0, j'ai utilisé la variable `pinOffset` qui contient une valeur de décalage pour définir la position de départ pour une boucle `for` afin de commander la première LED et toutes les autres.

Un tableau bidimensionnel possède au sens figuré une dimension spatiale de plus, passant ainsi quasiment d'une droite unidimensionnelle à une surface.

Figure 10-5 ▶
Tableau bidimensionnel



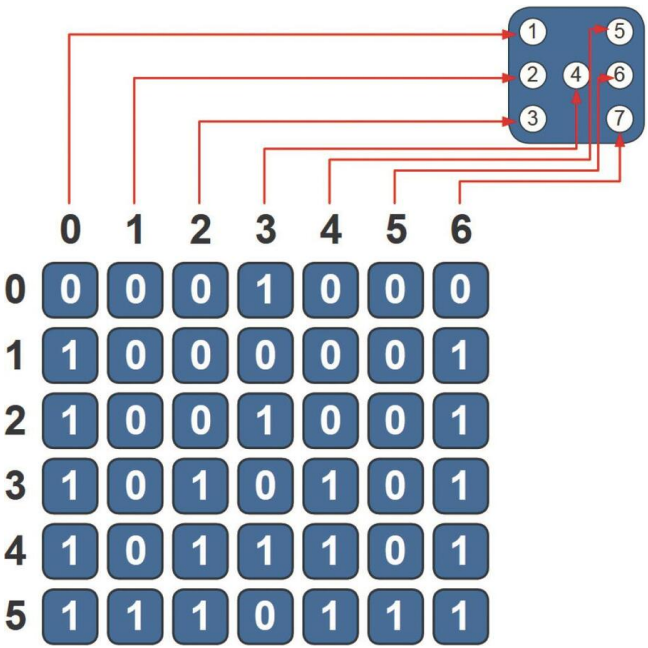
Il se comporte de la même manière que pour trouver une pièce sur un jeu d'échecs. On la localise plus facilement grâce à des coordonnées : par exemple Dame sur D1, D indiquant la colonne et 1 la rangée. Le tableau présenté ici dispose de 6 × 7 = 42 éléments. Déclaration et initialisation se font comme d'habitude. Seule la paire de crochets étant rajoutée pour la nouvelle dimension.

```
int pips[6][7] = {{0, 0, 0, 1, 0, 0, 0},  // Nombre sorti 1
                  {0, 0, 1, 0, 0, 0, 1},   // Nombre sorti 2
                  {0, 0, 1, 1, 0, 0, 1},   // Nombre sorti 3
                  {1, 0, 1, 0, 1, 0, 1},   // Nombre sorti 4
                  {1, 0, 1, 1, 1, 0, 1},   // Nombre sorti 5
                  {1, 1, 1, 0, 1, 1, 1}};  // Nombre sorti 6
```

La première valeur `[6]` entre crochets indique le nombre de lignes, le deuxième `[7]` le nombre de colonnes. La double paire de crochets permet aussi d'accéder à un élément :

```
pips[ligne][colonne]
```

Vous pouvez ainsi procéder ligne par ligne et lire les valeurs de LED correspondantes pour y accéder. La figure suivante montre l'affectation des différentes valeurs.



◀ Figure 10-6
Affectation des valeurs des colonnes du tableau aux LED correspondantes

Peut-être trouvez-vous bizarre que le graphique commence par 0 et finisse par 5 à la place de 6, alors qu'on ne peut pas faire 0 avec un dé. La réponse est simple. Ce ne sont pas les points du dé qui sont énumérés, mais l'index du tableau. Rappelez-vous que l'index commence toujours à `0` et présente

donc un décalage numérique de -1 par rapport aux points du dé. Voici maintenant un petit sketch qui affiche les contenus du tableau bidimensionnel sur le moniteur série :

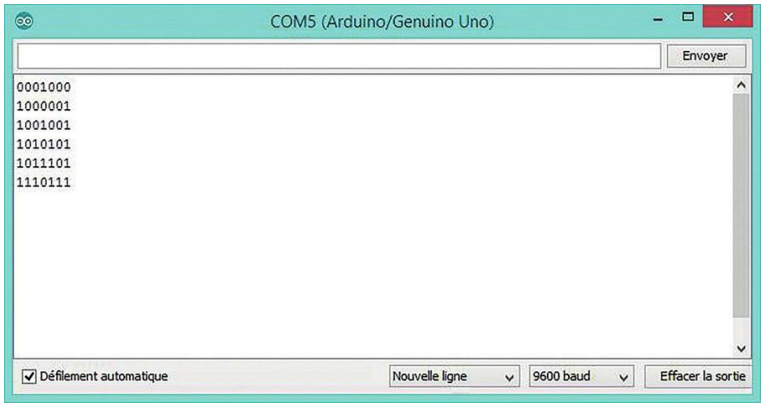
```
int pips[6][7] = {{0, 0, 0, 1, 0, 0, 0},  // Nombre sorti 1
                  {1, 0, 0, 0, 0, 0, 1},   // Nombre sorti 2
                  {1, 0, 0, 1, 0, 0, 1},   // Nombre sorti 3
                  {1, 0, 1, 0, 1, 0, 1},   // Nombre sorti 4
                  {1, 0, 1, 1, 1, 0, 1},   // Nombre sorti 5
                  {1, 1, 1, 0, 1, 1, 1}};  // Nombre sorti 6

void setup() {
  Serial.begin(9600);
  for(int row = 0; row < 6; row++){
    for(int col = 0; col < 7; col++)
      Serial.print(pips[row][col]);
    Serial.println();
  }
}

void loop() { /* ... */ }
```

Il s'agit ici de deux boucles `for` imbriquées. La boucle extérieure, qui contient la variable de contrôle `row` (ligne), commence à compter à partir de sa valeur de départ `0`. Vient ensuite la boucle intérieure, qui commence elle aussi par la valeur `0` de sa variable de contrôle `col` (colonne). La boucle intérieure doit cependant avoir fini de traiter toutes ses valeurs pour que la boucle extérieure incrémente la sienne.

IMBRICATION DE BOUCLES

Dans le cas de boucles imbriquées l'une dans l'autre, le traitement se fait de l'intérieur vers l'extérieur. Autrement dit, la boucle intérieure doit avoir exécuté tous ses passages avant que la boucle extérieure ne compte un de plus et que la boucle intérieure ne poursuive avec ses passages. Le cycle continue jusqu'à ce que toutes les boucles aient été traitées.

La figure 10-7 présente le contenu du tableau affiché sur le moniteur série.



◀ Figure 10-7
Contenu du tableau affiché ligne par ligne sur le moniteur série

Comparez ce résultat à l'initialisation du tableau : ils coïncident. Passons maintenant à l'analyse du code proprement dite. La fonction `setup` a encore pour tâche d'initialiser les différentes broches :

```
void setup() {
  for(int i = 0; i < 7; i++)
    pinMode(pin[i], OUTPUT);
  pinMode(buttonPin, INPUT);
}
```

Les broches pour commander les LED, broches programmées comme `OUTPUT` dans la fonction `setup`, sont également regroupées dans un tableau. Il n'y a qu'au bouton-poussoir, qui est relié à une entrée numérique, qu'une variable normale est affectée. La tâche principale est encore exécutée par la fonction `loop` :

```
void displayPips(int value) {
  for(int i = 0; i < 7; i++)
    digitalWrite(i + pinOffset, (pips[value - 1][i] == 1)?HIGH:LOW);
  delay(WAITTIME);
}

void loop() {
  if(digitalRead(buttonPin) == HIGH)
    displayPips(random(1, 7)); // Générer un nombre aléatoire
                               // entre 1 et 6
}
```

Quand le bouton-poussoir est enfoncé, la fonction `displayPips` est appelée. Un chiffre aléatoire compris entre 1 et 6 lui est transmis comme argument. Voyons maintenant de plus près le mode d'exploitation de la fonction.

Il s'agit essentiellement d'une boucle `for`, qui commande les différentes LED correspondant au chiffre transmis. Supposons qu'un 4 soit sorti : la fonction reçoit cette valeur comme argument. La boucle `for` commence son travail. Elle commande les broches et détermine le niveau `HIGH/LOW` nécessaire pour la LED en question :

```
for(int i = 0; i < 7; i++)
    digitalWrite ( + pinOffset, (pips[ value - 1][ i]  == 1)?HIGH:LOW);
```

Broche de la LED — Niveau HIGH/LOW

La variable `pinOffset` a pour valeur `2` et établit que la première broche à traiter se trouve à cette place. La première broche, portant le numéro `0`, est **RX** et la deuxième, portant le numéro 1, est **TX**. Ces deux broches sont en principe à éviter. La boucle `for` commençant par la valeur `0`, la valeur de `pinOffset` lui est rajoutée. Mais revenons à notre exemple, dans lequel un `4` est sorti. La boucle `for` traite la 4e ligne du tableau pour déterminer les niveaux `HIGH/LOW` nécessaires. Mais cette valeur doit être diminuée de 1 du fait que l'on commence par la valeur d'index `0`.

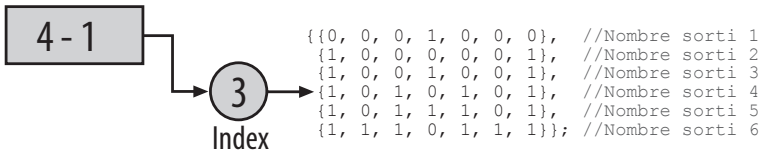


Figure 10-8 ▶ Sélection de l'élément du tableau pertinent pour un nombre préalablement sorti

La ligne sélectionnée dans le tableau comporte les valeurs 1, 0, 1, 0, 1, 0, 1 qui sont traitées une à une par la boucle `for`. Ceci est amorcé par l'expression suivante :

```
(pips[Value - 1][i] == 1)?HIGH:LOW)
```

Celle-ci vérifie que les valeurs sont bien `1` ou `0`. Le niveau `HIGH` est appliqué si c'est `1` et le niveau `LOW` si c'est `0`. Les LED correspondant au nombre sorti sont ainsi activées ou désactivées. Tant que le bouton-poussoir est maintenu enfoncé, un nouveau nombre est déterminé et les LED clignotent toutes très vite l'une derrière l'autre. Une fois le bouton-poussoir relâché, le dernier nombre reste affiché. La constante `WAITTIME` permet de régler la vitesse à laquelle les nombres changent quand le bouton-poussoir est enfoncé, soit ici 20 ms.

Vous vous souvenez du tableau unidimensionnel ? Nous avons vu qu'il est inutile d'écrire la dimension du tableau entre les crochets si celui-ci est initialisé aussitôt dans la même ligne. Vous vous dites alors peut-être qu'en

utilisant un tableau bidimensionnel, le compilateur déduirait des valeurs transmises les dimensions que doit avoir le tableau.

L'idée n'est pas mauvaise mais vous obtiendriez une erreur. En effet, si vous omettez toutes les indications sur la taille du tableau et écrivez

```
int pips[][] = {{0, 0, 0, 1, 0, 0, 0},   // Nombre sorti 1
                {1, 0, 0, 0, 0, 0, 1},   // Nombre sorti 2
                {1, 0, 0, 1, 0, 0, 1},   // Nombre sorti 3
                {1, 0, 1, 0, 1, 0, 1},   // Nombre sorti 4
                {1, 0, 1, 1, 1, 0, 1},   // Nombre sorti 5
                {1, 1, 1, 0, 1, 1, 1}};  // Nombre sorti 6
```

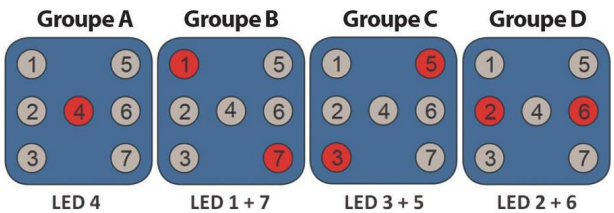
le compilateur renâcle, comme vous le constatez. Le message d'erreur dit, pour résumer, que dans le cas d'un tableau multidimensionnel, toutes les limites, hormis la première, doivent être indiquées. Vous pouvez donc écrire la ligne suivante :

```
int pips[][7] = ...
```

Le compilateur acceptera ce code.

Que pouvons-nous encore améliorer ?

Il est presque toujours possible d'améliorer ou de simplifier les choses. Il vous suffit de prendre un peu de recul et de considérer le projet dans son ensemble. Ne vous creusez pas trop la tête. Les idées viennent souvent quand on est occupé à autre chose. Revenons à notre dé. Si vous regardez les différents points d'un dé pour différentes valeurs, vous remarquerez peut-être quelque chose. Retournez pour ce faire au tableau « Quelle LED s'allume pour quel nombre ? » Question : les huit LED s'allument-elles toutes indépendamment les unes des autres ? Ou se peut-il que certaines forment un groupe ? Question idiote, non ? C'est le cas, bien évidemment : la figure suivante montre les différents groupes.

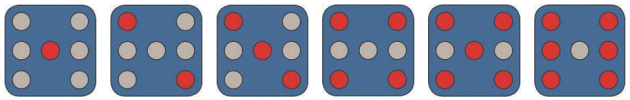


◀ **Figure 10-9**
Groupes de LED sur le dé électronique

Pris séparément, le groupe A et le groupe B sont utilisables, ce qui est moins le cas pour les groupes C et D. Quoi qu'il en soit, les configurations souhaitées sont générées par un groupe ou une combinaison de plusieurs

groupes. Voyons maintenant lequel ou lesquels des groupes est ou sont concerné(s) par quels points du dé :

Tableau 10-3 ▶ Points du dé et groupes de LED

Dé	
Groupe A	✓ ✓ ✓
Groupe B	✓ ✓ ✓ ✓ ✓
Groupe C	✓ ✓ ✓
Groupe D	✓

Ainsi, nous pouvons contrôler les LED avec 4 lignes de commandes au lieu de 7.

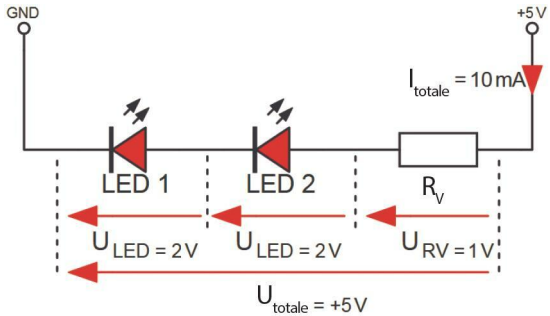
Nous devons interconnecter deux LED dans les groupes B, C, et D. Dans le montage n° 3, nous avions calculé la résistance série pour une LED rouge. Relisez-le si besoin. Si plusieurs LED doivent être commandées, il faut les brancher en série. Il y a environ 2 V aux bornes d'une seule LED rouge, autrement dit la résistance série doit faire chuter 3 V. Deux LED étant ici branchées l'une derrière l'autre, il est possible de déterminer ce qui suit pour la chute de tension aux bornes de la résistance série R_V :

$$U_{RV} = U_{totale} - U_{LED1} - U_{LED2} = +5\ \text{V} - 2\ \text{V} - 2\ \text{V} = 1\ \text{V}$$

1 V doit donc être « absorbé » par la résistance série R_V pour que 2 V subsistent aux bornes de chaque LED. Pour ce qui est du courant, qui circule de la même façon dans tous les composants (rappelez-vous comment le courant se comporte dans un montage en série), je le fixe à 10 mA (10 mA = 0,01 A). On obtient donc les valeurs suivantes dans la formule pour calculer la résistance série :

$$R_V = \frac{U_{totale} - U_{LED1 + LED2}}{I} = \frac{5\ \text{V} - 4\ \text{V}}{0{,}01\ \text{A}} = 100\ \Omega$$

Le circuit ressemble à ceci :

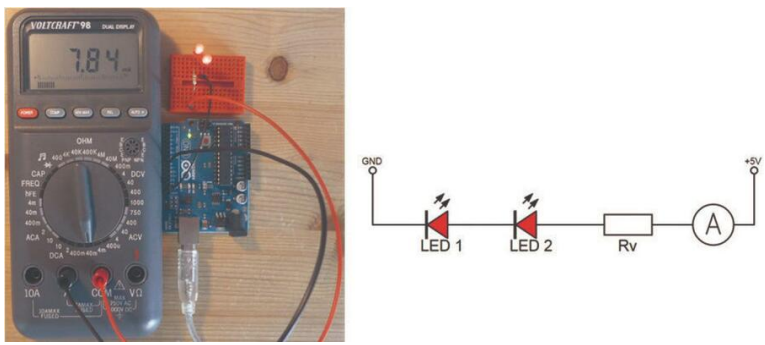


◀ Figure 10-10
Deux LED avec une résistance série

ATTENTION AU SENS DES LED !

Veillez à ce que les deux LED soient dans le même sens, sinon pas d'allumage. L'anode de la LED 1 est reliée à la cathode de la LED 2.

Ici aussi, j'ai vérifié le calcul de manière pratique pour m'assurer que tout est également en ordre.



◀ Figure 10-11
Mesure du courant sur le circuit de commande avec deux LED et une nouvelle résistance

Le courant de 7,84 mA est absolument correct et encore inférieur à la prescription de 10 mA maxi. Deux LED ayant naturellement besoin du double de tension d'alimentation par rapport à une seule, la résistance série doit être plus faible pour que la luminosité des deux LED, soit la même que celle d'une seule LED. Vous pouvez bien sûr utiliser la même résistance de 330 Ω pour tous les groupes A à D, ce qui signifie toutefois en théorie que la luminosité du groupe A sera plus forte avec une seule LED que le reste des groupes.

Passons maintenant à la programmation. Par quoi commencer ? Je vous suggère de revenir au tableau 10-3 et de voir s'il s'en dégage une systématique établissant quel groupe de LED doit être commandé, à quel moment et pour quelles configurations de points du dé. Procédez étape par étape

et observez un groupe après l'autre. Vous pouvez les traiter complètement à part l'un de l'autre, car la logique de commande se charge ensuite de les rassembler pour un affichage en commun des véritables points du dé. Je vous montre encore une fois de manière simplifiée le groupe A du tableau 10-3.

Dé	1	2	3	4	5	6
Groupe A	✓		✓		✓	

Encore un indice : qu'est-ce que les nombres 1, 3 et 5 ont en commun ? Ce sont tous des nombres impairs !

Formulation pour commander le groupe A

Commander le groupe A si le nombre aléatoire déterminé est impair.

Passons maintenant au groupe B. Voici l'extrait correspondant du tableau 10-3 :

Dé	1	2	3	4	5	6
Groupe B		✓	✓	✓	✓	✓

Que constatez-vous ici ? Tous les nombres sont concernés sauf le 1.

Mais à quoi pourrait bien ressembler une formulation que le microcontrôleur comprendrait sans problème ? Une description quelque peu maladroite donnerait ceci : commander le groupe B si le nombre est 2 ou 3 ou 4 ou 5 ou 6. Cherchez là encore le point commun et vous pourrez raccourcir fortement la formulation.

Formulation pour commander le groupe B

Commander le groupe B si le nombre aléatoire déterminé est supérieur à 1.

Voyons maintenant le groupe C. Tous les nombres supérieurs à 3 sont concernés.

Dé	1	2	3	4	5	6
Groupe C				✓	✓	✓

Formulation pour commander le groupe C

Commander le groupe C si le nombre aléatoire déterminé est supérieur à 3.

Passons pour finir au groupe D.

Dé	1	2	3	4	5	6
Groupe C						✓

Formulation pour commander le groupe D

Commander le groupe D si le nombre aléatoire déterminé est égal à 6.

Passons maintenant à la programmation. Vous verrez que cette solution est beaucoup plus simple que l'utilisation d'un tableau. Mais il faut suivre mentalement quelques pistes jusqu'au bout avant de s'apercevoir que 4 broches au lieu de 7 sont nécessaires pour commander les LED. Cela permet cependant d'aborder cette thématique par le côté ludique. Voici le code du sketch pour commander le dé électronique avec moins de lignes de commande :

```
#define WAITTIME 20
int GroupA = 8;      // LED 4
int GroupB = 9;      // LED 1 + 7
int GroupC = 10;     // LED 3 + 5
int GroupD = 11;     // LED 2 + 6
int buttonPin = 13;  // Bouton-poussoir sur broche 13

void displayPips(int value) {
  // Éteindre tous les groupes
  digitalWrite(GroupA, LOW);
  digitalWrite(GroupB, LOW);
  digitalWrite(GroupC, LOW);
  digitalWrite(GroupD, LOW);
  // Commande de tous les groupes
  if(value%2 != 0) // La valeur est-elle impaire ?
    digitalWrite(GroupA, HIGH);
  if(value > 1)
    digitalWrite(GroupB, HIGH);
  if(value > 3)
    digitalWrite(GroupC, HIGH);
   if(value == 6)
    digitalWrite(GroupD, HIGH);
  delay(WAITTIME); // Ajouter une courte pause
}

void setup() {
  pinMode(GroupA, OUTPUT);
  pinMode(GroupB, OUTPUT);
  pinMode(GroupC, OUTPUT);
  pinMode(GroupD, OUTPUT);
}

void loop() {
  if(digitalRead(buttonPin) == HIGH)
    displayPips(random(1, 7)); // Générer un nombre entre 1 et 6
}
```

Peut-être pensez-vous que nous avons oublié quelque chose ! En effet, nous avons programmé les broches pour les groupes A à D comme sortie, mais nous n'avons pas défini le bouton-poussoir comme entrée.

Effectivement, je n'ai pas programmé cette entrée sur la broche 13 comme entrée. Vous avez raison sur ce point. Mais je n'ai pas non plus oublié puisque toutes les broches numériques sont définies comme entrée de manière standard et n'ont donc pas besoin d'être explicitement programmées en cas d'utilisation appropriée. Vous pouvez bien entendu le faire partout dans votre sketch, car cela aide certainement à comprendre.

Examinons la ligne qui détermine si la valeur est impaire. L'opérateur `%` (opérateur modulo) permet de calculer le reste d'une division. Si le nombre est divisible par 2, il s'agit d'un nombre pair, le reste de la division étant dans ce cas toujours égal à `0`. La ligne

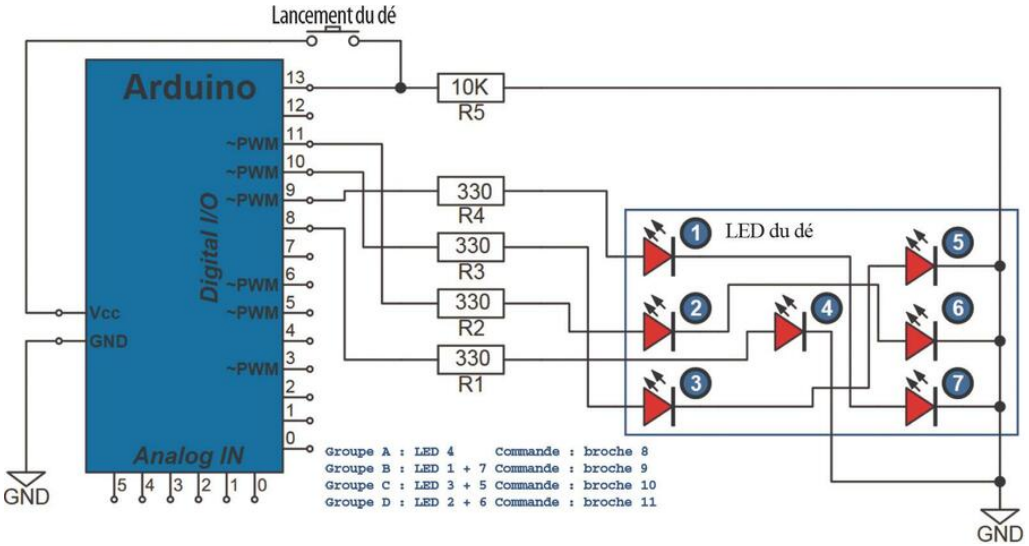
```
if(value%2 != 0)
```

me permet cependant de demander si le reste est différent de `0` pour ainsi commander le groupe A.

Encore une remarque avant d'en venir au circuit : cela ne change pas grand-chose si, au lieu de la résistance série de 100 Ω calculée pour les groupes B à D, vous utilisez ici les anciennes résistances de 330 Ω. La luminosité semble être la même, mais ce n'est qu'approximatif.

Figure 10-12 ▼
Carte Arduino commandant les 7 LED du dé par groupe de LED

La figure suivante montre qu'il faut moins de résistances série pour les LED que dans le montage précédent.



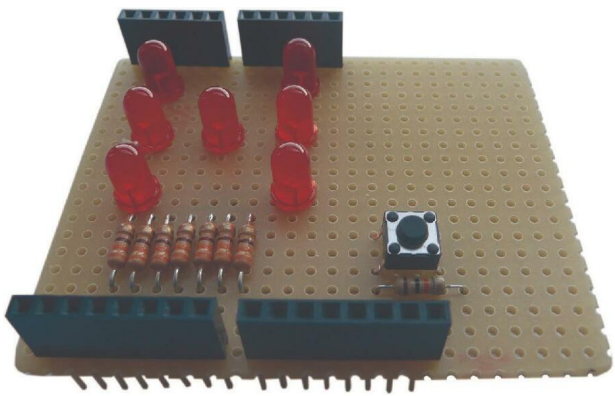
Problèmes courants

Si les LED ne se mettent pas à clignoter après avoir appuyé sur le bouton-poussoir ou si les points du dé qui s'affichent sont bizarres, voire incohérents, 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 ?
- Pas de court-circuit éventuel ?
- Les différentes LED sont-elles correctement branchées ? La polarité est-elle correcte ?
- Les résistances ont-elles 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.

Montage du dé électronique sur une platine

Il est possible de créer soi-même ses platines afin de les utiliser ensuite comme extensions de la carte Arduino. Je vais vous présenter la solution que j'ai imaginée pour le dé électronique.

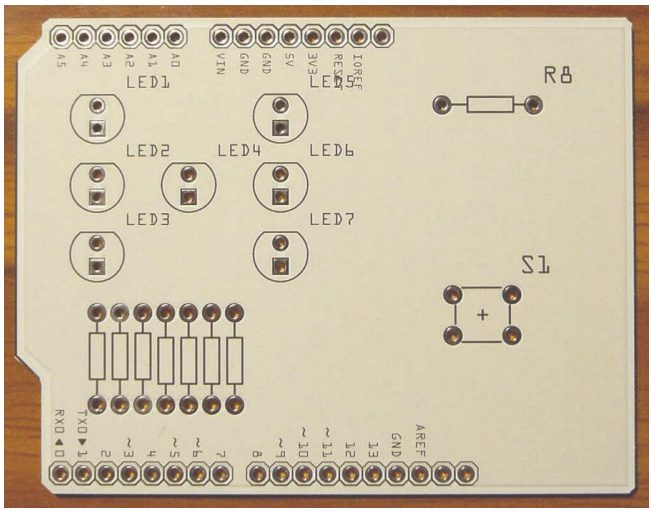


◀ **Figure 10-13**
Dé électronique sur une platine

Qu'en pensez-vous ? C'est une jolie extension dont la réalisation et la mise en œuvre m'ont bien amusé. Plein d'idées me sont venues au cours du montage ! Ça donne envie d'aller encore plus loin ! Vous aussi, si vous le souhaitez, vous pouvez réaliser ce circuit imprimé comme un vrai profes-

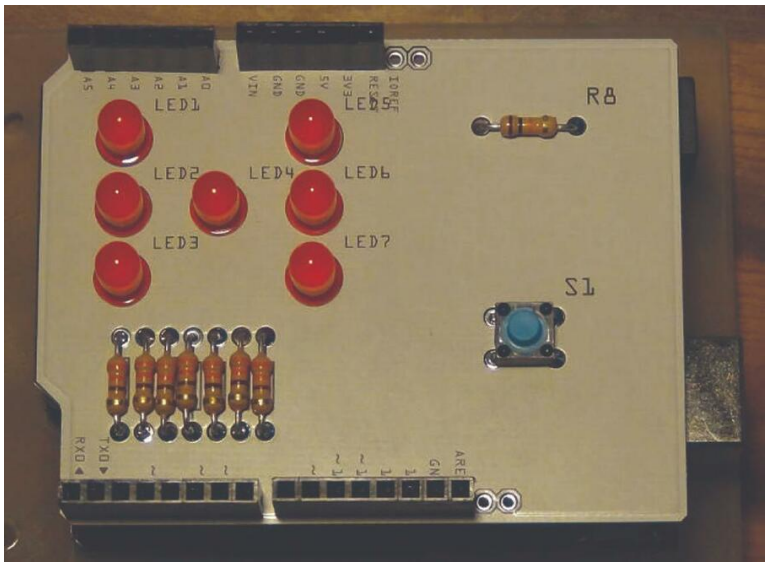
sionnel. La figure 10-14 montre la platine, ou shield, terminée, mais encore sans ses composants.

Figure 10-14 ▶
Dé électronique sur une platine après la production



C'est une véritable platine qui ressemble déjà beaucoup à un shield, avec tous ses marquages et ses trous forés. Il ne me reste plus qu'à y fixer les composants afin de voir ce que ça donne.

Figure 10-15 ▶
Circuit imprimé du dé électronique



Je suis très fier du résultat !

Exercice complémentaire

L'objet de cet exercice est déjà un peu plus délicat. Vous vous souvenez sûrement du registre à décalage 74HC595 avec ses 8 sorties. Essayez de réaliser un circuit ou de programmer un sketch qui commande un dé électronique au moyen du registre à décalage. Combien de broches numériques économisez-vous avec cette variante ? Est-ce un avantage par rapport à la réalisation avec des groupes de LED ?

Qu'avez-vous appris ?

- Vous avez appris dans ce montage comment déclarer et initialiser un tableau bidimensionnel et comment accéder aux différents éléments de ce tableau.
- Vous savez comment afficher des contenus de variables dans le moniteur série pour vérifier l'exactitude de ces valeurs. Vous pouvez ainsi rechercher une erreur et analyser le code en cas de comportement incorrect. Vous devez cependant être sûr que le circuit est correctement câblé, sinon vous chercherez dans le code source une erreur qui, en réalité, se situe au niveau du matériel. Cela vous évitera de perdre du temps et vous épargnera peut-être même une crise de nerfs.
- Vous avez appris comment calculer une résistance série pour deux LED montées en série, de manière à ce que la luminosité demeure pratiquement inchangée.

Montage

11

Des détecteurs de lumière

Dans ce montage, nous allons connecter un capteur à la carte Arduino afin de recevoir une valeur mesurée qui sera ensuite analysée. Nous sommes soumis à d'innombrables influences extérieures auxquelles nous ne pouvons pas nous soustraire, comme

- la température
- la luminosité
- l'humidité de l'air
- la pression atmosphérique
- le champ magnétique
- ...

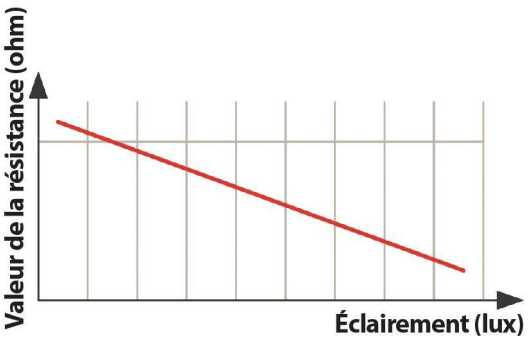
pour n'en citer que quelques-unes.

Maintenant que l'on sait prélever une valeur mesurée, il n'y a plus qu'un pas à franchir pour influencer le courant en fonction de la grandeur mesurée. Le moyen le plus simple pour y parvenir est d'utiliser une résistance. Ce n'est évidemment pas une résistance fixe, comme nous en avons utilisé jusqu'ici, mais une résistance variable. Si seule la mesure de la luminosité nous intéresse, c'est très facile à réaliser au moyen d'une photorésistance, également nommée résistance photosensible (en anglais, *Light Dependent Resistor* (LDR). Il s'agit d'un semiconducteur, dont la résistance dépend de la lumière. Plus la quantité de lumière atteignant la LDR est élevée, plus la résistance est faible. Notre circuit doit commander dix LED en fonction de la luminosité. Plus la luminosité est élevée, plus le nombre de LED allumées est important. Le circuit ressemble à celui du séquenceur de lumière, à ceci près que les différentes LED ne sont pas commandées l'une après l'autre par une boucle, mais par une logique qui évalue la luminosité sur la résistance photosensible.

La résistance variable

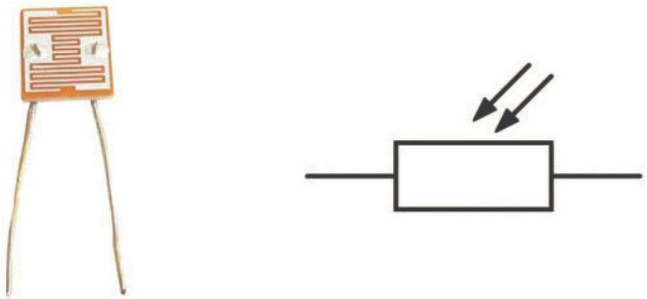
La résistance fixe n'a désormais plus aucun secret pour vous. Mais, dans ce montage, nous allons procéder autrement. Il existe une résistance dont la valeur varie en fonction de la luminosité ambiante, comme je l'ai déjà évoqué. C'est la LDR. Sa courbe caractéristique traduit son comportement en fonction de la variation de son éclairement.

Figure 11-1 ▶
Courbe caractéristique d'une LDR



Plus la lumière incidente est forte, plus la résistance de la LDR est faible. Sur la figure suivante, vous pouvez voir une LDR et son symbole. C'est ce symbole qui est utilisé pour représenter une LDR dans un schéma.

Figure 11-2 ▶
Une LDR et son symbole

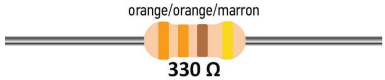


La plage de résistance de la LDR dépend du matériau employé et présente approximativement une résistance dans l'obscurité comprise entre 1 et 10M. Une intensité d'éclairement de 1 000 lux (lx) environ engendre une résistance comprise entre 75 et 300 Ω. Le *lux* est l'unité de mesure de l'éclairement. Voyons d'abord la liste des composants.

Composants nécessaires

Ce montage ne nécessite pas grand-chose et il peut même se passer de composants supplémentaires. En effet, la carte Arduino comporte une LED

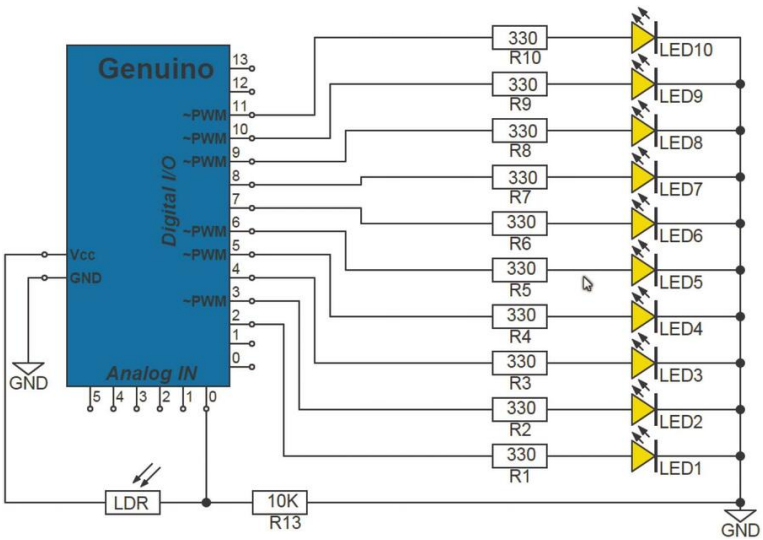
qui est désignée par la lettre *L*. Toutefois, j'aimerais compléter ce montage par quelques composants que nous réutiliserons pour d'autres montages – et que vous utiliserez aussi pour vos projets personnels.

Composant	
7 LED jaunes	
10 résistances de 330 Ω	
1 résistance de 10 kΩ	
1 LDR	

◀ Tableau 11-1
Liste des composants

Schéma

La seule inconnue éventuelle de ce schéma est le diviseur de tension, mais nous y reviendrons très vite.

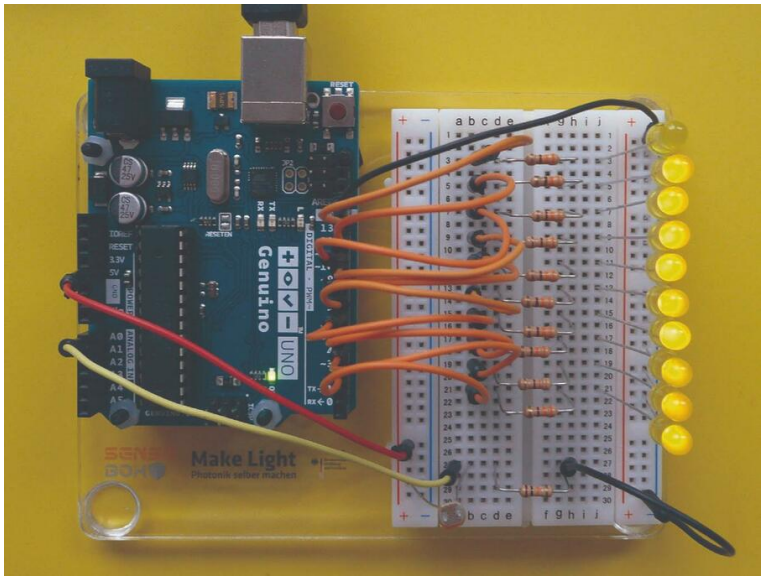


◀ Figure 11-3
Circuit de mesure de l'énergie lumineuse

Réalisation du circuit

La réalisation du circuit vous demandera un peu de travail, car vous devez connecter dix LED, ainsi que leurs résistances série.

Figure 11-4 ▶
La réalisation du circuit sur une petite plaque de prototypage



Sketch Arduino

Voici le code du sketch :

```
int pin[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13}; // Tableau des broches
int analogPin = A0; // Broche de l'entrée analogique

// Fonction pour commander les LED
void controlLEDs(int value){
  int bargraphValue = map(value, 0, 1023, 0, 9);
  for(int i = 0; i < 10; i++)
    digitalWrite(pin[i], (bargraphValue >= i)?HIGH:LOW);
}

void setup() {
  for(int i = 0; i < 10; i++)
    pinMode(pin[i], OUTPUT); // Toutes les broches comme sorties
}

void loop() {
  controlLEDs(analogRead(analogPin));
}
```

UTILISATION D'ALIAS

À la place de nombres entiers pour les entrées analogiques, vous pouvez aussi utiliser les alias `A0`, `A1`, `A2`, `A3`, `A4` et `A5`. Le code est sans doute un peu plus parlant.

Revue de code

Pour simplifier la configuration et la commande des LED au moyen des broches numériques, nous utiliserons un tableau qui ici se nomme `pin`. Le tableau est déclaré et initialisé dans la première ligne, les valeurs étant placées entre deux accolades. Toutefois, comme la carte Arduino ne peut travailler en interne qu'avec des valeurs binaires, cela impose de convertir les valeurs analogiques d'une quelconque façon. Cette tâche est réalisée par un convertisseur numérique/analogique, (ou convertisseur A/D). La carte Arduino dispose d'un convertisseur A/D à six entrées qui sont numérotées de A0 à A5. L'illustration ci-dessous présente le connecteur comportant les entrées analogiques.

◀ Figure 11-5
Connecteur à entrées analogiques

Chacune de ces entrées peut supporter des tensions comprises entre 0 et 5 V, la tension existante étant convertie en un nombre entier compris entre 0 et 1023.

Vous trouverez de plus amples informations sur les entrées analogiques aux adresses suivantes :

https://www.arduino.cc/en/Tutorial/AnalogInputPins

https://www.arduino.cc/en/Reference/AnalogRead

Pour ce montage, nous utiliserons uniquement l'entrée A0. Cette configuration est stockée dans la variable `analogPin`. La fonction `setup` programme toutes les broches numériques comme sorties en se référant au tableau. La commande est effectuée à l'aide d'une fonction qui a été programmée dans ce but. Elle se nomme `controlLEDs` et est exécutée en continu dans la fonction `loop`. La valeur analogique mesurée A0 est transmise comme argument. Intéressons-nous maintenant à fonction programmée :

```
void controlLEDs(int value){
  int bargraphValue = map(value, 0, 1023, 0, 9);
  for(int i = 0; i < 10; i++)
    digitalWrite(pin[i], (bargraphValue >= i)?HIGH:LOW);
}
```

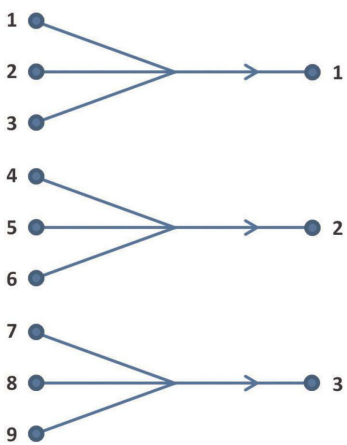
Elle possède un paramètre qui stocke la valeur mesurée sur l'entrée analogique. Ce code me donne l'occasion d'introduire une nouvelle instruction très intéressante qui permet de transposer un domaine de valeur dans un autre domaine. J'ai déjà précisé que la tension mesurée était convertie en nombre entier compris entre 0 et 1023. Avec l'instruction ou la fonction `map`, il est très facile de convertir le domaine des valeurs d'entrée, qui s'étend de 0 à 1023, en un domaine de valeurs de sortie allant de 0 à 9.

Vous trouverez de plus amples informations sur l'instruction `map` à l'adresse suivante :

https://www.arduino.cc/en/Reference/Map

La figure suivante illustre le processus, les domaines d'entrée et de sortie ayant été réduits pour plus de clarté. Au lieu de convertir 1024 en 10, j'ai converti 9 en 3.

Figure 11-6 ▶
Conversion de 9 à 3

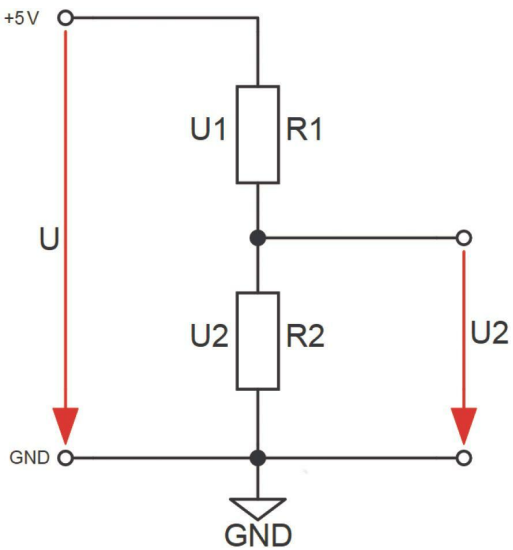


La représentation d'un grand domaine de valeurs par un domaine plus petit nécessite une conversion qui entraîne obligatoirement une diminution de la résolution. Les valeurs d'entrées sont représentées comme suit :

- Valeur1 de 1 à 3 : représentée par 1

- Valeur1 de 4 à 6 : représentée par 2
- Valeur1 de 7 à 9 : représentée par 3

Les dix LED sont raccordées aux sorties numériques 2 à 11. Prenons un circuit de base qui comporte deux résistances. La disposition suivante correspond à celle d'un *diviseur de tension*. Nous en avons déjà vu une similaire pour le calcul de la résistance série d'une diode.



◀ Figure 11-7
Diviseur de tension avec deux résistances fixes

Comme son nom l'indique, un diviseur de tension divise une tension en un certain nombre de parties. Sur le côté gauche, la tension d'origine U est fournie à deux résistances R1 et R2 qui sont montées en série, c'est-à-dire l'une derrière l'autre. La mesure de tension de sortie U2 se trouve sur le côté droit et elle est parallèle à la résistance R2. Ce circuit comporte un nouveau symbole pour la masse, qui est désignée par les lettres GND *(Ground)*. Il se situe sur le bord inférieur du circuit. Les tensions et les résistances sont liées par la relation de proportionnalité suivante :

$$\frac{\textit{tension totale}}{\textit{tension partielle}} = \frac{\textit{résistance totale}}{\textit{résistance partielle}}$$

Si nous appliquons cette formule à notre circuit, nous obtenons :

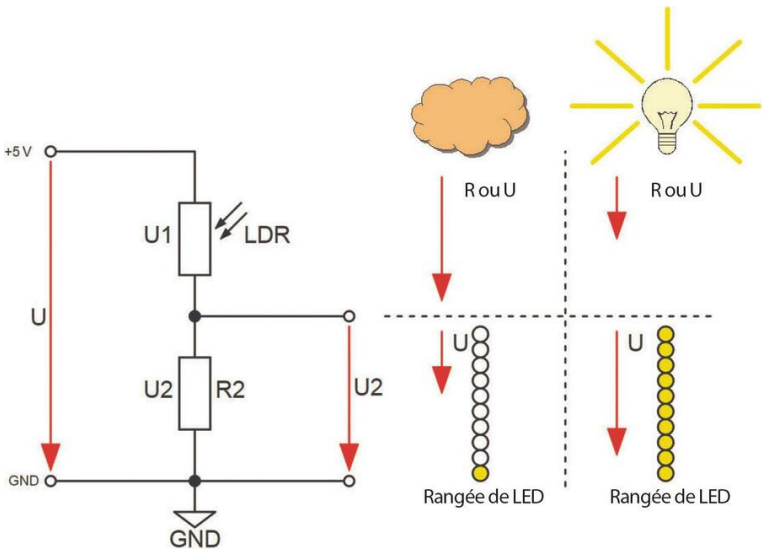
$$\frac{U}{U_2} = \frac{R_1 + R_2}{R_2}$$

Pour calculer la tension de sortie, il faut simplement isoler U2 de cette équation :

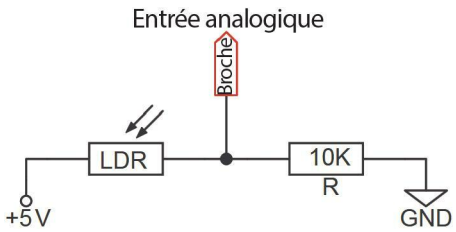
$$U_2 = U \cdot \frac{R_2}{R_1 + R_2}$$

Si nous remplaçons maintenant l'une des deux résistances fixes par une résistance variable, que nous utiliserons comme photorésistance pour les besoins de notre montage, nous obtenons alors le circuit suivant avec le diviseur de tension modifié :

Figure 11-8 ▶
Diviseur de tension avec une LDR et une résistance fixe



Si la tension la plus élevée se trouve aux bornes de la plus grande résistance, la tension la plus élevée doit se trouver à l'entrée analogique quand la LDR s'assombrit ? Servez-vous du schéma précédent. À droite du circuit, j'ai représenté les rapports de tension. La longueur de la flèche correspond à la valeur de tension. Si le ciel est couvert, la résistance ou plutôt la tension est élevée sur la LDR. Mais si le soleil brille, résistance et tension sont faibles. Comme le diviseur de tension est alimenté sous 5 V, cela signifie qu'il ne reste que la différence de tension aux bornes de la résistance R2. Cette tension est mesurée entre l'entrée analogique et la masse. Voici, pour mémoire, le raccordement de la photorésistance sur le circuit :



Nous avons affaire à un diviseur de tension.

Devenons communicatifs

Il est certes intéressant à mes yeux d'observer le cycle des LED sous différentes conditions de lumière, mais le déroulement chronologique reste difficile à voir sur une longue période. Aussi voudrais-je vous présenter ici un projet qui vous plaira sûrement, puisqu'il est agréable à regarder. Le langage de programmation *Processing* s'impose quand il s'agit de générer des graphiques.

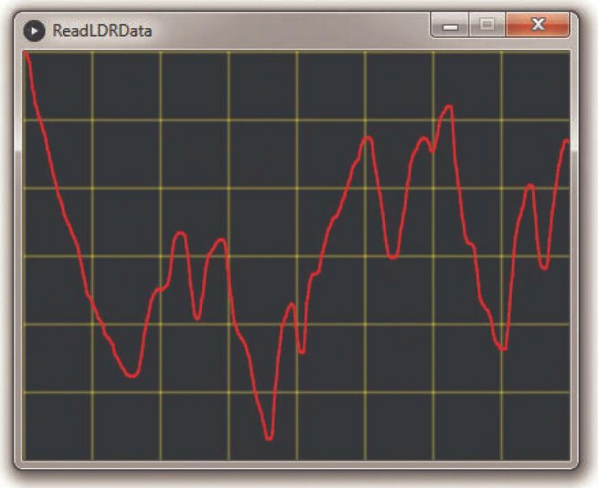
Vous trouverez l'environnement de développement pour le langage de programmation Processing sur le lien suivant :

https://processing.org/

Le côté pratique de la chose est que, tout comme pour Arduino, il suffit de décompresser le fichier téléchargé dans un répertoire. Aucune installation n'est à faire. Ce qui prendrait un temps fou avec des langages de programmation tels que *C++* ou *C#* sera plus vite fait et sera moins fastidieux avec Processing.

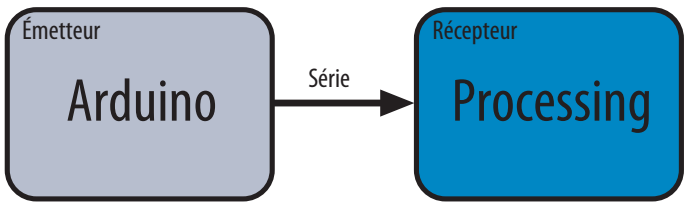
Je vous montre d'emblée le résultat dans la fenêtre graphique de Processing pour que vous sachiez à quoi vous attendre.

Figure 11-9 ▶
Courbe des valeurs de l'énergie lumineuse dans la fenêtre graphique de Processing



Les valeurs affichées sont actualisées en permanence, la courbe évoluant de droite à gauche dans la fenêtre. Les nouvelles valeurs apparaissent à droite et les anciennes valeurs disparaissent à gauche de la fenêtre.

Mais comment deux langages de programmation différents font pour échanger des données ? Une base commune leur permettant de se comprendre doit être définie. L'interface série vous dit sûrement déjà quelque chose. Tout langage de programmation ou presque a dans son vocabulaire des instructions pour envoyer ou recevoir par cette interface. Notre exemple montre un émetteur et un récepteur. La communication est unidirectionnelle, autrement dit se fait dans un seul sens. Certes, l'interface est capable de communiquer presque simultanément dans les deux sens, mais nous nous contenterons d'un seul.



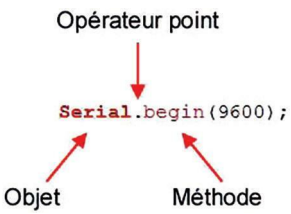
Tout ce que votre carte Arduino doit faire, c'est enregistrer les valeurs mesurées et envoyer les données via l'interface série. Plus vite à dire qu'à faire ?! Non, pas du tout, car la plupart du travail de calcul se fait du côté de Processing. Voyons d'abord du côté de l'émetteur ce qu'Arduino doit faire.

Arduino l'émetteur

Côté matériel, il ne vous faut que le diviseur de tension avec sa LDR et sa résistance de 10K fixe, connecté à l'entrée analogique broche `0`, pour envoyer les valeurs de luminosité à l'interface série. Le sketch est le suivant :

```
void setup() {
  Serial.begin(9600); // Initialise le port série

}

void loop() {
  Serial.println(analogRead(0));  // Transmet la valeur au port série
  delay(10);                      // Courte pause (très important !!!)
}
```

Voyons maintenant ce que ce bout de code donne. Dans la fonction `setup`, l'interface série est préparée pour la transmission. Vous avez eu vos premiers contacts avec la programmation orientée objet lors de la création de votre propre bibliothèque dans le montage n° 8. L'interface série est considérée comme étant un objet logiciel nommé `Serial`. Vous disposez de quelques méthodes, que nous entendons maintenant utiliser.



La méthode pour initialiser l'interface a pour nom `begin` et reçoit une valeur qui détermine la vitesse de la transmission. Il s'agit dans notre cas de 9600. L'unité de mesure est ici le baud, le nombre indiquant la cadence. 1 baud signifie 1 changement d'état par seconde.

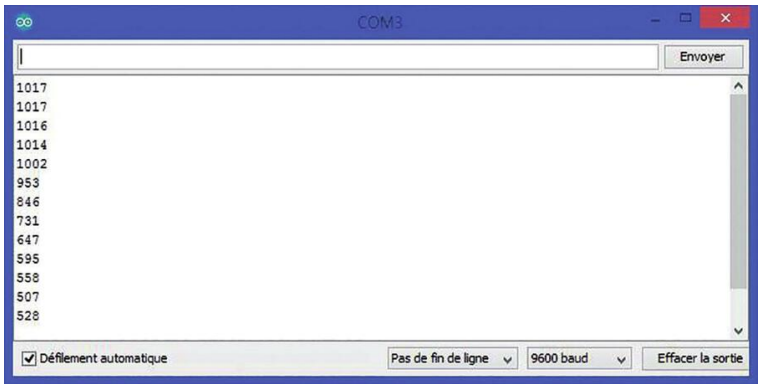
La deuxième méthode que nous utiliserons se nomme `println`. Elle envoie la valeur qui lui a été transmise à l'interface série. Il s'agit de la valeur mesurée sur la broche analogique `0` dans notre bout de sketch. L'interrogation de la broche analogique et la transmission à l'interface ont lieu continuellement dans la fonction `loop`.

TAUX DE TRANSFERT

Pour que la communication entre émetteur et récepteur puisse se faire, le réglage du taux de transfert doit être le même sur les deux stations.

Vous pouvez suivre la transmission des valeurs déterminées en temps réel, de préférence en ouvrant le moniteur série de l'environnement de développement.

Figure 11-10 ▶
Édition des données dans le moniteur série



Cela fonctionne évidemment aussi avec n'importe quel autre programme de terminal ayant accès à l'interface série. Pensez ici aussi à régler le taux de transfert tout en bas à droite de la fenêtre.

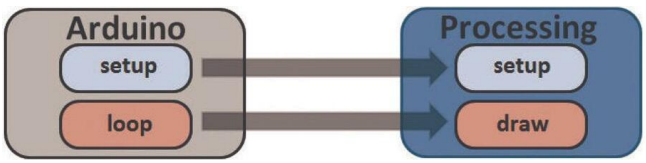
Processing le récepteur

Venons en maintenant au programme en question, chargé principalement de représenter graphiquement les valeurs reçues. Le code est assez conséquent, mais voici quand même une courte description pour que vous ne soyez pas perdu.

```
import processing.serial.*;

Serial mySerialPort;
int xPos = 1;
int serialValue;
int[] yPos;

void setup() {
  size(400, 300);
  println(Serial.list());
  mySerialPort = new Serial(this, Serial.list()[0], 9600);
  mySerialPort.bufferUntil('\n');
  // Définir la couleur d'arrière-plan
  background(0);
  yPos = new int[width];
}

void draw() {
  background(0);
```

```
  stroke(255, 255, 0, 120);
  for(int i=0; i < width; i+=50)
    line(i, 0, i, height);
  for(int i=0; i < height; i+=50)
    line(0, i, width, i);

  stroke(255, 0, 0);
  strokeWeight(1);
  int yPosPrev = 0, xPosPrev = 0;
  println(serialValue);
  // Décaler les valeurs du tableau vers la gauche
  for(int x = 1; x < width; x++)
    yPos[x-1] = yPos[x];
  // Joindre les nouvelles coordonnées de la souris
  // à l'extrémité droite du tableau
  yPos[width - 1] = serialValue;
  // Affichage du tableau
  for(int x = 0; x < width; x++){
    if(x >0)
      line(xPosPrev, yPosPrev, x, yPos[x]);
    xPosPrev = x;       // Stockage de la dernière position x
    yPosPrev = yPos[x]; // Stockage de la dernière position y
  }
}

void serialEvent(Serial mySerialPort) {
  String portStream = mySerialPort.readString();
  float data = float(portStream);
  serialValue = height - (int)map(data, 0, 1023, 0, height);
}
```

Processing comprend également deux fonctions principales, qui ressemblent à celles d'Arduino.



◀ Figure 11-11
Correspondance entre deux fonctions principales

La fonction `setup` est appelée une seule fois elle aussi en début de sketch et sert à initialiser des variables. La fonction `draw` est une boucle sans fin semblable à la fonction `loop` dans Arduino, qui doit son nom au fait qu'elle sert à dessiner (en anglais, *drawing*) les éléments graphiques dans la fenêtre d'édition. Pour pouvoir traiter l'interface série dans Processing, vous devez écrire la ligne :

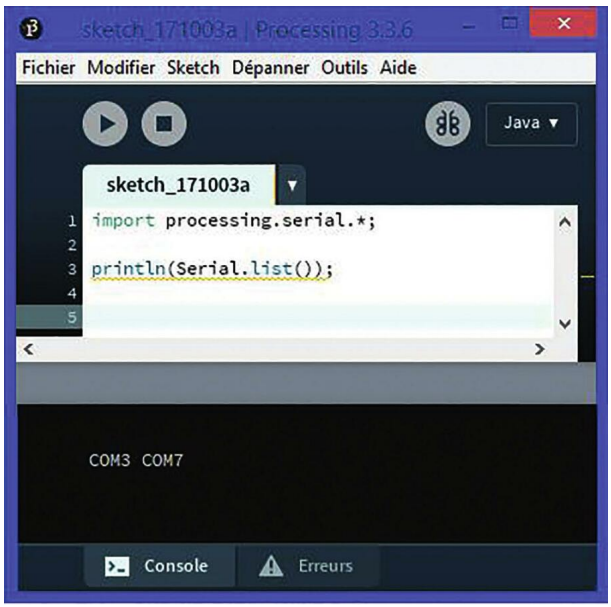
```
import processing.serial.*;
```

qui permet d'importer un paquet en langage Java. Eh oui, Processing est un langage basé sur Java, contrairement à Arduino qui, lui, utilise C ou C++. Mais ces langages ont une syntaxe très similaire, si bien que la programmation dans Processing ne semblera pas compliquée à ceux qui connaissent bien C ou C++. Si vous écrivez la ligne :

```
println(Serial.list());
```

Processing vous donne une liste de toutes les interfaces série disponibles. L'affichage dans la fenêtre de messagerie ressemble alors à ceci.

Figure 11-12 ▶
Affichage des ports COM disponibles



Elle indique que deux ports sériels sont disponibles. Arduino utilisant le premier port *COM3* correspondant à la première entrée de la liste `[0]`, cet index est reporté dans la ligne ci-après :

```
mySerialPort = new Serial(this, Serial.list()[0], 9600);
```

Si, sur votre ordinateur, le port série utilisé n'est pas le premier de la liste, il faut alors remplacer dans la ligne précédente `Serial.list()[0]` par `"COMn"`, où `n` est le numéro du port série effectivement utilisé par Arduino. Un nouvel objet sériel est ainsi généré, et instancié avec le mot-clé `new` dans Processing. La valeur `9600` apparaît une fois de plus, elle doit correspondre à celle qui est dans le sketch Arduino. Tous les éléments graphiques tels que trame de fond et courbe sont alors dessinés dans la fonction `draw`. Les valeurs Arduino transmises s'accumulent dans la fonction `serialEvent` et

sont stockées dans la variable `serialValue`. Cette variable sert à dessiner la courbe dans la fonction `draw`.

UTILISATION D'UN MONITEUR SÉRIE

Si vous avez ouvert un programme de terminal, par exemple un moniteur série, pour visualiser les valeurs envoyées par Arduino, vous aurez des problèmes si vous démarrez en même temps l'affichage graphique des valeurs dans Processing. Le port COM concerné est en effet exclusivement utilisé par le programme de terminal, interdisant tout accès supplémentaire par un autre programme. Fermez par conséquent le moniteur série avant de démarrer Processing pour évaluer les données.

Problèmes courants

Si les différentes LED ne réagissent pas aux modifications des conditions lumineuses ou si aucun changement du tracé de la courbe n'est à observer par la suite dans la fenêtre d'édition, il peut y avoir plusieurs raisons.

- Vos fiches de raccordement sur la plaque d'essais correspondent-elles vraiment au circuit ?
- Pas de court-circuit éventuel ?
- Les résistances ont-elles bien les bonnes valeurs ?
- Toutes les LED sont-elles correctement polarisées ?
- Vérifiez encore une fois l'exactitude du code du sketch côté Arduino et côté Processing.
- Ouvrez le moniteur série dans l'IDE Arduino pour vous assurer que des valeurs différentes sont transmises à l'interface série, lorsque les conditions lumineuses varient. Dans le code de Processing, vous pouvez ajouter la ligne `println(serialValue)` de manière à ce que les valeurs transmises (pour peu qu'elles le soient) s'affichent également dans la fenêtre de messagerie.
- Vérifiez que l'interface série utilisée n'est pas bloquée par un autre processus et que seul Processing y accède.

Exercice complémentaire

Créez un sketch Arduino faisant clignoter régulièrement toutes les LED concernées quand par exemple un certain seuil de flux lumineux est franchi, pour vous avertir qu'un état critique est maintenant atteint et qu'une crème solaire avec indice de protection 75+ doit être utilisée.

Qu'avez-vous appris ?

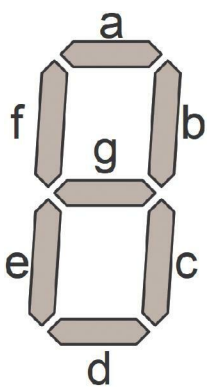
- Vous savez comment interroger une entrée analogique à laquelle est reliée une résistance photosensible (LDR) avec l'instruction `analogRead`.
- Un diviseur de tension sert à diviser la tension appliquée à ses bornes dans un rapport déterminé. Nous avons utilisé cette propriété pour amener à l'entrée analogique une tension fonction de l'intensité lumineuse.
- Vous savez comment échanger des données entre deux programmes au moyen de l'interface série. L'émetteur est ici la carte Arduino et le récepteur un sketch Processing reproduisant visuellement les données reçues sous forme de courbe.

12 L'afficheur sept segments

Pour visualiser des états logiques (vrai ou faux) ou des données (`14`, `2.5`, `"Hello User"`) sous une forme quelconque, il nous faut commander les LED dans un premier temps et revenir au moniteur série dans un deuxième temps. Il existe en électronique d'autres éléments d'affichage que les LED, l'afficheur sept segments étant l'un d'eux. Comme son nom l'indique, cet afficheur se compose de sept segments qui, disposés d'une certaine manière, peuvent représenter des chiffres et, dans une moindre mesure, des signes.

Qu'est-ce qu'un afficheur sept segments ?

La figure 12-1 présente un tel afficheur de manière schématisée.



◀ Figure 12-1
Afficheur 7 segments

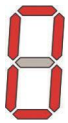
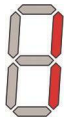
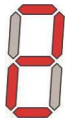
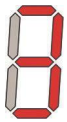
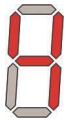
On voit que chaque segment est pourvu d'une petite lettre. L'ordre n'est pas primordial, mais la forme montrée ici s'est imposée et a été adoptée pratiquement partout. Aussi l'utiliserons-nous également toujours sous cette forme. Si maintenant nous commandons les différents

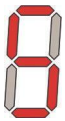
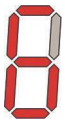
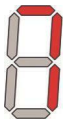
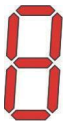
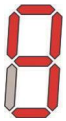
segments avec habileté, nous pouvons afficher des chiffres allant de 0 à 9. On peut aussi afficher des lettres, nous y reviendrons plus tard. Votre quotidien est sûrement rempli de ces afficheurs sept segments sans que vous n'y ayez jamais prêté attention. Faites un tour en ville et vous verrez à quel point ils sont courants. Voici d'ailleurs une petite liste des possibilités d'utilisation :

- l'affichage des prix dans les stations-service (toujours en hausse hélas !) ;
- l'affichage de l'heure sur certains bâtiments ;
- l'affichage de la température ;
- les montres digitales ;
- les tensiomètres médicaux ;
- les thermomètres numériques.

Le tableau 12-1 indique une fois pour toutes, en vue de la programmation, quels sont les segments à allumer pour chacun des chiffres.

Tableau 12-1 ▶ Commande des sept segments

Afficheur	a	b	c	d	e	f	g
	1	1	1	1	1	1	0
	0	1	1	0	0	0	0
	1	1	0	1	1	0	1
	1	1	1	1	0	0	1
	0	1	1	0	0	1	1

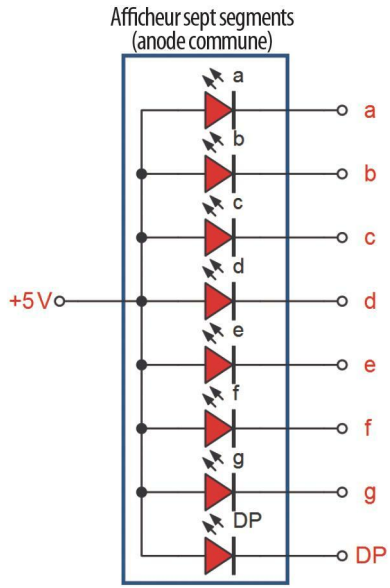
Afficheur	a	b	c	d	e	f	g
	1	0	1	1	0	1	1
	1	0	1	1	1	1	1
	1	1	1	0	0	0	0
	1	1	1	1	1	1	1
	1	1	1	1	0	1	1

Le chiffre `1` dans ce tableau ne signifie pas forcément niveau `HIGH`, mais c'est la commande de l'allumage du segment concerné. Celle-ci peut se faire soit avec le niveau `HIGH` que nous connaissons (+5 V résistance série incluse), soit avec un niveau `LOW` (0 V). Vous voulez peut-être savoir maintenant en fonction de quoi on choisit une commande. Cela dépend en fait du type de l'afficheur sept segments. Deux approches sont possibles :

- la cathode commune ;
- l'anode commune.

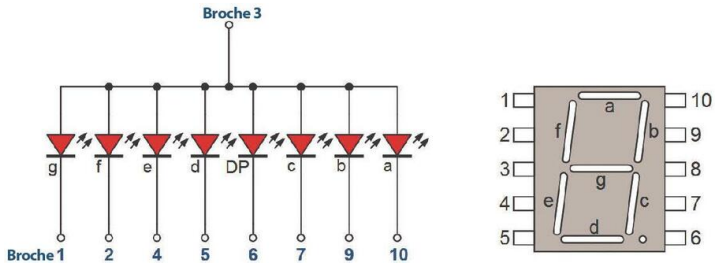
En cas de cathode commune, toutes les cathodes des diverses LED de l'afficheur sept segments sont réunies en interne et reliées à la masse à l'extérieur. Les différents segments sont commandés par des résistances série dûment raccordées au niveau `HIGH`. Notre exemple porte cependant sur un afficheur sept segments avec anode commune. Ici, c'est exactement le contraire : toutes les anodes des diverses LED sont reliées entre elles en interne et raccordées au niveau `LOW` à l'extérieur. Les segments sont commandés par des résistances série correctement dimensionnées, en passant par les différentes cathodes des LED qui sont accessibles à l'extérieur.

Figure 12-2 ▶
Afficheur 7 segments avec anode commune



Sur la figure suivante, vous pouvez voir que toutes les anodes de l'afficheur sept segments sont reliées à la tension d'alimentation +5 V. Les cathodes sont reliées par la suite aux sorties numériques de votre carte Arduino et pourvues des différents niveaux de tension conformes au tableau de commande. Nous utilisons pour notre essai un afficheur sept segments avec anode commune de type *SA 39-11 GE*. La figure suivante illustre le brochage de cet afficheur.

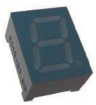
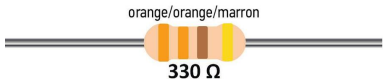
Figure 12-3 ▶
Commande de l'afficheur sept segments de type SA 39-11 GE



Le graphique de gauche montre les broches utilisées de l'afficheur sept segments, et le graphique de droite le brochage du type utilisé. *DP* est la forme abrégée de *Decimal Point* (point décimal).

Composants nécessaires

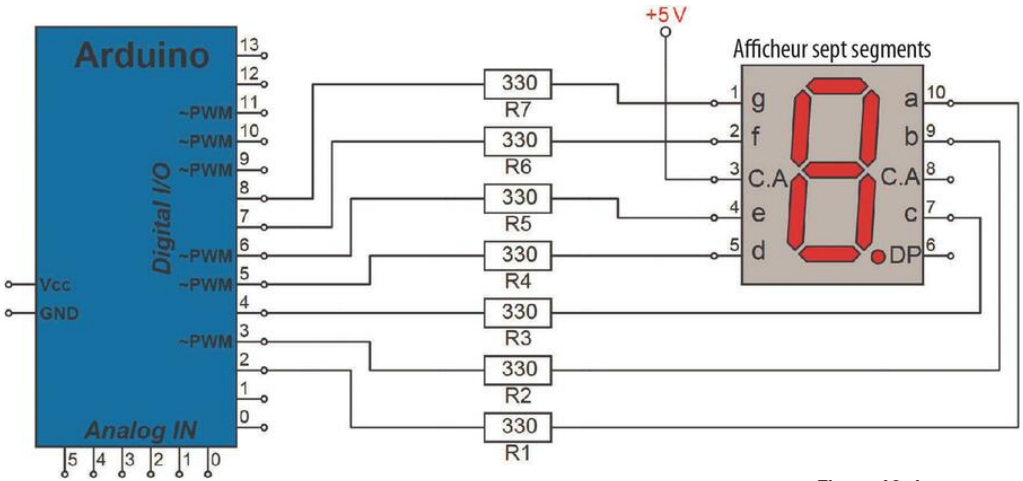
Ce montage nécessite les composants suivants.

Composant	
1 afficheur sept segments (par exemple de type SA 39-11 GE avec anode commune)	
7 résistances de 330 Ω	

◀ **Tableau 12-2**
Liste des composants

Schéma

Le circuit ressemble à celui du séquenceur de lumière. Toutefois, les différentes sorties numériques ne sont évidemment pas commandées successivement, mais dans une combinaison précise, puisqu'elles doivent composer un chiffre particulier :



▲ **Figure 12-4**
Carte Arduino commandant l'afficheur 7 segments

Réalisation du circuit

La réalisation du circuit montre les sept résistances sur le bord supérieur et l'afficheur 7 segments sur le bord inférieur de la plaque d'essais.

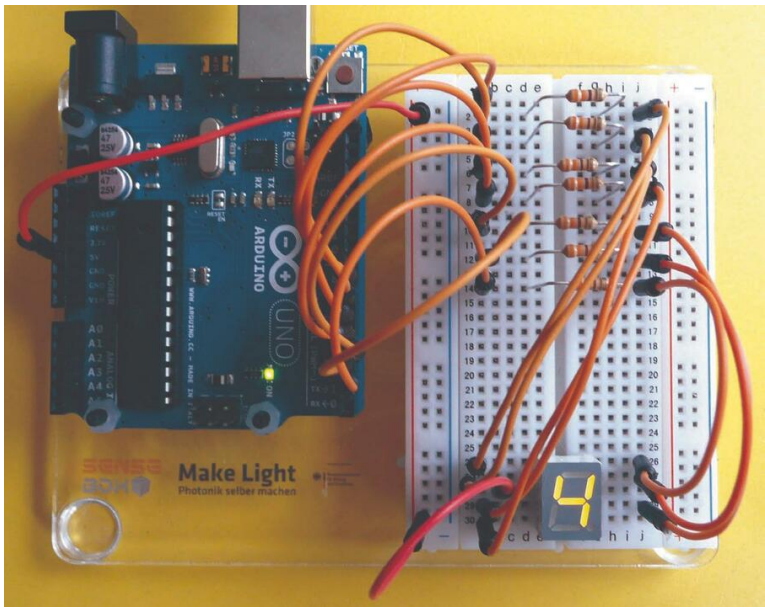


Figure 12-5 ▶
Réalisation du circuit de l'afficheur 7 segments

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

Pour faciliter la commande des sept segments, le sketch fait ici aussi appel à un tableau.

```
int segments[10][7] = {{1, 1, 1, 1, 1, 1, 0},  // 0
                       {0, 1, 1, 0, 0, 0, 0},  // 1
                       {1, 1, 0, 1, 1, 0, 1},  // 2
                       {1, 1, 1, 1, 0, 0, 1},  // 3
                       {0, 1, 1, 0, 0, 1, 1},  // 4
                       {1, 0, 1, 1, 0, 1, 1},  // 5
                       {1, 0, 1, 1, 1, 1, 1},  // 6
                       {1, 1, 1, 0, 0, 0, 0},  // 7
                       {1, 1, 1, 1, 1, 1, 1},  // 8
                       {1, 1, 1, 1, 0, 1, 1}}; // 9
int pinArray[] = {2, 3, 4, 5, 6, 7, 8};

void setup() {
  for(int i = 0; i < 7; i++)
    pinMode(pinArray[i], OUTPUT);
}

void loop() {
```

```
  for(int i = 0; i < 10; i++) {
    for(int j = 0; j < 7; j++)
      digitalWrite(pinArray[j], (segments[i][j]==1)?LOW:HIGH);
    delay(1000); // Pause de 1 seconde
  }
}
```

Revue de code

Un tableau bidimensionnel s'impose d'emblée pour stocker les informations sur les segments à allumer pour chaque chiffre de `0` à `9`. Ces valeurs sont définies dans la variable globale `segments` en début de sketch :

```
int segments[10][7] = {{...},
                       ...
                      {...}};
```

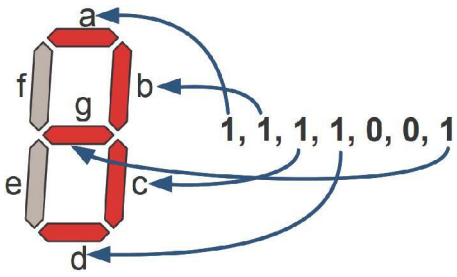
Le tableau comprend 10 × 7 cases mémoire, le contenu de chacune pouvant être obtenu par les coordonnées :

```
segments[x][y]
```

La coordonnée `x` sert pour tous les chiffres de `0` à `9` (soit 10 cases mémoire), et la coordonnée `y` pour tous les segments de `a` à `g` (soit 7 cases mémoire). On détermine par exemple les segments à allumer du chiffre `3` en écrivant la ligne :

```
segments[3][y]
```

les résultats pour la variable *y* allant de `0` à `6` étant obtenus par une boucle `for`. Les données des segments sont alors celles de la figure suivante.



Pour plus de clarté, j'ai uniquement dessiné les flèches des segments à allumer.

Peut-être vous demandez-vous pourquoi il y a un 1 là où il devrait y avoir une mise à la masse dans le tableau des segments, alors que nous avons dit que ce type d'afficheur sept segments disposait d'une anode commune.

En fait, j'ai dit qu'un `1` ne voulait pas forcément dire niveau `HIGH`, mais simplement que le segment en question devait être allumé. Dans le cas d'un afficheur sept segments à cathode commune, on commande l'allumage du segment souhaité avec le niveau `HIGH`, tandis que dans le cas d'un afficheur sept segments à anode commune, on le commande avec le niveau `LOW`. On écrit ainsi la ligne suivante :

```
digitalWrite(pinArray[j], (segments[i][j]==1)?LOW:HIGH);
```

Si l'information est un `1`, `LOW` est alors transmis comme argument à la fonction `digitalWrite`. Sinon, c'est `HIGH`. Le segment correspondant s'allume si c'est `LOW`, et se voit géré de manière à rester éteint si c'est `HIGH`. Notre sketch affiche tous les chiffres de `0` à `9` au rythme d'une seconde. Le code suivant est utilisé pour ce faire :

```
for(int i = 0; i < 10; i++) {
  for(int j = 0; j < 7; j++)
    digitalWrite(pinArray[j], (segments[i][j]==1)?LOW:HIGH);
  delay(1000); // Pause de 1 seconde
}
```

La boucle extérieure avec la variable de contrôle `i` sélectionne dans le tableau le chiffre à afficher tandis que la boucle intérieure avec la variable `j` sélectionne les segments à allumer.

Sketch amélioré

Les divers segments d'un chiffre étaient commandés jusqu'ici au moyen d'un tableau bidimensionnel, la première dimension servant à sélectionner le chiffre désiré, et la deuxième les différents segments. Le sketch suivant va nous permettre de tout faire avec un tableau unidimensionnel. Comment ? C'est simple puisque bits et octets n'ont déjà plus de secret pour vous. L'information de segment doit maintenant tenir dans une seule valeur. Quel type de donnée s'impose ici ? Nous avons affaire à un afficheur sept segments, et à un point décimal que nous laisserons de côté pour l'instant. Cela fait donc 7 bits, qui tiennent idéalement dans un seul octet de 8 bits. Chaque bit est simplement affecté à un segment et tous les segments nécessaires peuvent être commandés avec un seul octet. J'en profite pour vous montrer comment initialiser directement une variable par le biais d'une combinaison de bits :

```
void setup() {
  Serial.begin(9600);
  byte a = B10001011;     // Déclarer + initialiser la variable
  Serial.println(a, BIN); // Afficher en tant que valeur binaire
  Serial.println(a, HEX); // Afficher en tant que valeur hexadécimale
  Serial.println(a, DEC); // Afficher en tant que valeur décimale
}

void loop(){ /* vide */ }
```

La ligne décisive est bien sûr la suivante :

```
byte a = B10001011;
```

Ce qui est remarquable pour ne pas dire génial là-dedans, c'est le fait que le préfixe `B` permet de représenter une combinaison de bits qui sera affectée à la variable située à gauche du signe `=`. Cela simplifie les choses quand par exemple vous connaissez une combinaison de bits et souhaitez la sauvegarder. Il vous faudrait sinon convertir la valeur binaire en valeur décimale avant de sauvegarder. Cette étape intermédiaire n'est ici plus nécessaire.

Peut-être vous interrogez-vous sur le type de donnée `byte` qui est bien un nombre entier. Or type de donnée et nombres entiers sont bien composés de chiffres allant de 0 à 9. Alors pourquoi maintenant peut-on commencer par la lettre B et la faire suivre d'une combinaison de bits ? Ou s'agit-il d'une chaîne de caractères ?

Effectivement, le type de donnée `byte` est un type de nombre entier. Vous avez raison sur ce point. Là où vous avez tort, c'est sur le fait qu'il pourrait s'agir d'une chaîne de caractères. Celle-ci serait alors entre guillemets. Il s'agit en fait de tout autre chose. Aucune idée ? Je ne dirai qu'un mot : `#define`. Ça vous dit quelque chose ? Voyez plutôt. Il existe dans les tréfonds d'Arduino un fichier nommé `binary.h` qui se trouve dans le répertoire `...\Arduino\hardware\arduino\avr\cores\arduino`. Voici un court extrait de ce fichier, dont les nombreuses lignes n'ont pas toutes besoin d'être montrées.

```
#ifndef Binary_h
#define Binary_h

#define B0 0
#define B00 0
#define B000 0
#define B0000 0
#define B00000 0
#define B000000 0
#define B0000000 0
#define B00000000 0
#define B1 1
#define B01 1
#define B001 1
#define B0001 1
#define B00001 1
#define B000001 1
#define B0000001 1
#define B00000001 1
```

Ce fichier contient toutes les combinaisons de bits possibles pour les valeurs de 0 à 255, qui y sont définies en tant que constantes symboliques. Je me suis permis de retirer la ligne pour la valeur 139 (déconseillé, à moins de restaurer ensuite l'état initial !) pour voir comment le compilateur réagit. Voyez plutôt :

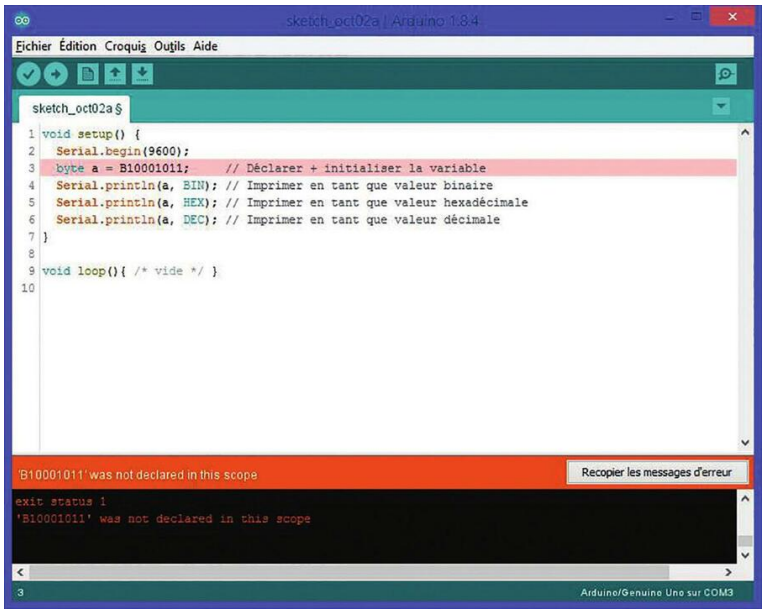


Figure 12-6 ▶
Message d'erreur intentionnelle

Le message d'erreur indique que le nom `B10001011` n'a pas été trouvé. Il me faut encore vous expliquer les lignes suivantes avant d'en revenir au projet :

```
Serial.println(a, BIN); // Afficher en tant que valeur binaire
Serial.println(a, HEX); // Afficher en tant que valeur hexadécimale
Serial.println(a, DEC); // Afficher en tant que valeur décimale
```

La fonction `println` peut accueillir, en plus de la valeur à afficher, un autre argument qui peut être indiqué séparé par une virgule. Je vous ai mis ici les trois plus importants. Vous en trouverez d'autres sur la page de référence des instructions Arduino sur Internet. Des explications figurent sous forme de commentaires derrière les lignes d'instructions. L'affichage dans le moniteur série est alors la suivante :

```
10001011
8B
139
```

Passons maintenant à la commande de l'afficheur sept segments au moyen du tableau bidimensionnel. Voici auparavant le sketch complet que nous allons analyser :

```
byte segments[10]= {B01111110,  // 0
                    B00110000,  // 1
                    B01101101,  // 2
                    B01111001,  // 3
                    B00110011,  // 4
                    B01011011,  // 5
                    B01011111,  // 6
                    B01110000,  // 7
                    B01111111,  // 8
                    B01111011}; // 9
int pinArray[] = {2, 3, 4, 5, 6, 7, 8};

void setup() {
  for(int i = 0; i < 7; i++)
    pinMode(pinArray[i], OUTPUT);
}

void loop() {
  for(int i = 0; i < 10; i++) { // Commande du chiffre
  for(int j = 6; j >= 0; j--){  // Interrogation des bits pour les segments
      digitalWrite(pinArray[6 - j], bitRead(segments[i], j) ==
1?LOW:HIGH);
    }
    delay(500); // Attendre une demi-seconde
  }
}
```

Sur la figure 12-7, on voit très bien quel bit est en charge de quel segment au sein de l'octet.

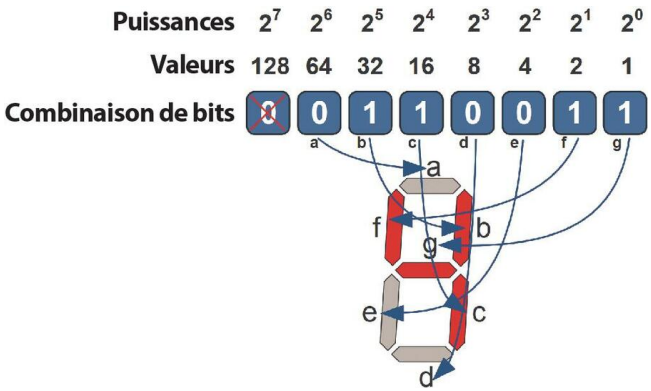


Figure 12-7 ▶
Un octet gère les segments de l'afficheur (ici par exemple pour le chiffre 4).

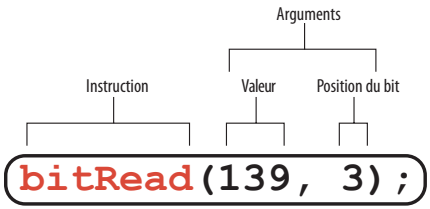
Ayant seulement sept segments à commander et ne tenant pas compte du point décimal, j'ai constamment donné au MSB (rappelez-vous : MSB = bit le plus significatif) la valeur `0` pour tous les éléments du tableau. Tout se joue bien entendu encore – et comment en serait-il autrement – à l'intérieur de la fonction `loop`. Jetons-y un coup d'œil :

```
void loop() {
  for(int i = 0; i < 10; i++) {   // Commande du chiffre
   for(int j = 6; j >= 0; j--){  // Interrogation des bits pour les segments
    digitalWrite(pinArray[6 - j], bitRead(segments[i], j) == 1?LOW:HIGH);
   }
   delay(500); // Attendre une demi-seconde
  }
}
```

La boucle extérieure `for` avec la variable de contrôle `i` commande encore les divers chiffres de 0 à 9. C'était déjà le cas dans la première solution. Le code est ensuite différent. La boucle intérieure `for` avec la variable de contrôle `j` est chargée de choisir le bit dans le chiffre sélectionné. Je commence du côté gauche par la position `6`, qui est en charge du segment *a*. Le tableau des broches gérant cependant la broche `8` pour le segment `g` à la position `6` de l'index, la commande doit se faire en sens inverse. On y parvient en soustrayant le nombre `6` puisque j'ai gardé tel quel le tableau des broches du premier exemple :

```
pinArray[6 - j]
```

Voici maintenant à une fonction intéressante, permettant de lire un bit déterminé dans un octet. Elle porte le nom `bitRead`.



◀ **Figure 12-8**
Instruction `bitRead`

Cet exemple donne le bit de la position `3` pour la valeur décimale `139` (binaire : 10001011). Le comptage commence pour l'index `0` au LSB (bit le moins significatif) du côté droit. La valeur renvoyée serait par conséquent un `1`. La ligne :

```
digitalWrite(pinArray[6 - j], bitRead(segments[i], j) == 1?LOW:HIGH);
```

permet de vérifier que la lecture du bit sélectionné renvoie bien un 1. Si c'est le cas, la broche sélectionnée est commandée avec le niveau `LOW`, autrement dit le segment s'allume. N'oubliez pas : *anode commune* ! Sauriez-vous expliquer la différence entre les deux solutions ?

Dans la première version avec le tableau bidimensionnel, le chiffre à afficher est sélectionné par la première dimension tandis que les segments à commander le sont par la deuxième. Cette information se trouve dans les différents éléments du tableau. Dans la deuxième version, le chiffre à afficher est également sectionné par la première dimension. S'agissant d'un tableau unidimensionnel, elle est cependant la seule dimension. Seulement, l'information pour commander les segments est contenue dans les diverses valeurs de l'octet. Ce qui était fait auparavant par la deuxième dimension est maintenant fait par les bits d'un octet.

Problèmes courants

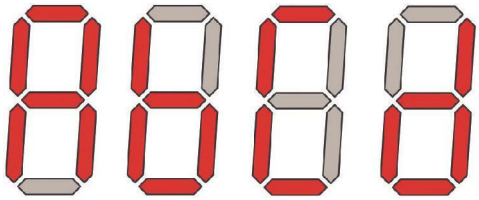
Si l'affichage ne correspond pas aux chiffres 1 à 9 ou si des combinaisons incohérentes s'affichent, vérifiez les choses suivantes.

- Vos fiches de raccordement sur la plaque correspondent-elles vraiment au circuit ?
- Pas de court-circuit éventuel ?

- Le code du sketch est-il correct ?
- Si des caractères incohérents s'affichent, il se peut que vous ayez interverti des lignes de commande. Vérifiez le câblage avec le schéma ou la fiche technique de l'afficheur sept segments.
- Le tableau des segments est-il initialisé avec les bonnes valeurs ?

Exercice complémentaire

Élargissez la programmation du sketch de telle sorte que certaines lettres puissent s'afficher à côté des chiffres 0 à 9. Ce n'est certes pas possible pour tout l'alphabet, donc à vous de trouver lesquelles pourraient convenir. La figure suivante vous fournit quelques exemples pour commencer.



Notez qu'il existe un nombre infini de déclinaisons d'afficheurs sept segments. L'affichage peut être de différentes couleurs, telles que :

- jaune ;
- rouge ;
- vert ;
- rouge très clair.
- Il faut bien entendu s'assurer du type de connexion avant d'acheter :
- l'anode commune ;
- la cathode commune.

Elles ont des tailles différentes. En voici deux proposées par le fournisseur Kingbright :

- type *SA-39* : hauteur des chiffres = 0,39" = 9,9 mm ;
- type *SA-56* : hauteur des chiffres = 0,56" = 14,2 mm.

Qu'avez-vous appris ?

- Dans ce montage, les principes de la commande d'un afficheur sept segments vous sont expliqués.
- L'initialisation d'un tableau vous permet de définir les différents segments de l'affichage pour pouvoir les commander à votre aise par la suite.
- Le fichier d'en-tête `binary.h` contient un grand nombre de constantes symboliques que vous pouvez utiliser dans votre sketch.
- Vous savez comment convertir un nombre à afficher dans une autre base numérique en ajoutant un deuxième argument (`BIN`, `HEX` ou `DEC`) à la méthode `println`.
- La fonction `bitRead` vous permet de lire l'état de certains bits d'un nombre.

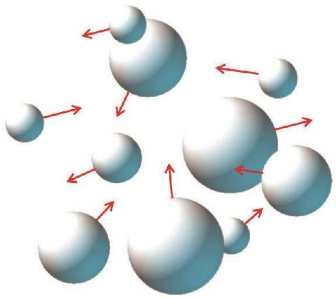
Montage 13

La température

Dans le montage n° 12, nous avions vu comment commander des afficheurs sept segments. Cette fois, nous irons encore plus loin. Vous avez certainement déjà vu ces thermomètres numériques qui affichent également l'heure devant une pharmacie, par exemple. Nous allons voir ici comment afficher la température ambiante à l'aide d'un afficheur sept segments. Mais avant cela, nous devons revoir quelques notions fondamentales.

Chaud ou froid ?

Nous vivons dans un monde ou plutôt dans un environnement composé de matières diverses. Ces dernières peuvent en principe présenter trois états dits *d'agrégation* en physique. Un tel état d'agrégation peut être solide, liquide ou gazeux et dépend souvent d'une grandeur physique appelée température. Mais que signifie la température et comment se fait-elle sentir ou plutôt comment peut-on la mesurer ? Toute matière est composée d'infimes particules appelées *atomes*. Ceux-ci sont composés d'un nuage d'*électrons* (charge : négative) et d'un noyau formé de *protons* (charge positive) et de *neutrons* (charge : nulle). Ce ne sont pas là les plus petites particules, mais elles suffiront à expliquer au moyen de notre exemple ce qu'est la température.



◀ **Figure 13-1**
Le mouvement des atomes

Ces infimes particules sont en perpétuel mouvement, errant apparemment sans but et dans des directions différentes. La température est donc un moyen de mesurer cette agitation thermique des atomes ou des molécules (assemblage de plusieurs atomes). Plus ils se déplacent rapidement, plus la probabilité est grande qu'ils entrent en collision. C'est alors que l'énergie cinétique se transforme en énergie calorifique. L'agitation thermique est donc un moyen de mesurer la température d'une matière.

Comment peut-on mesurer la température ?

On utilise des capteurs, qui convertissent la température mesurée en diverses valeurs de résistance ou de tension. Dans le cas le plus simple, on utilise des résistances *NTC* ou *PTC*. Elles manquent cependant de précision, et leur courbe caractéristique n'est pas forcément linéaire. De quel type de résistance s'agit-il et comment se comportent-elles en cas de fluctuation de la température ?

La résistance NTC

NTC signifie *Negative Temperature Coefficient* ou résistance à coefficient de température négatif. La résistance diminue avec la hausse de la température et conduit mieux le courant.

Figure 13-2 ▶
Résistance NTC

Sa forme ressemble à celle d'un condensateur céramique, ce qui peut parfois donner lieu à des confusions. Une inscription, par exemple 4K7, laisse pourtant présumer de la valeur de résistance, et l'appellation « Thermistor NTC 4K7 » permet de l'identifier formellement. Le symbole est le suivant :

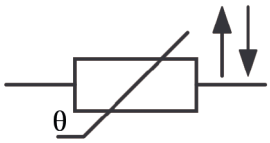


Figure 13-3
Symbole de la résistance NTC

Vous pouvez voir deux flèches à côté du symbole de la résistance. La première, à gauche, désigne la température et la deuxième, à droite, correspond à la résistance. Quand la température augmente, comme le signale la flèche pointée vers le haut, la résistance diminue, ce qui est indiqué par la flèche pointée vers le bas. Une NTC se reconnaît à sa courbe caractéristique.

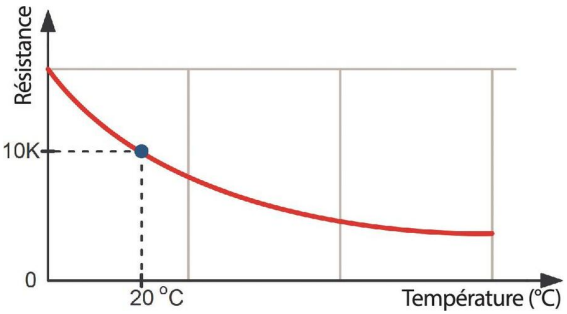


Figure 13-4
Courbe caractéristique d'une thermistance NTC

On remarque qu'il s'agit bel et bien d'une courbe et non d'une droite, ce qui est important pour la précision de la mesure de la température. La principale caractéristique de cette résistance est sa valeur de référence, R_{20}, qui est annoncée à une température ambiante de 20 °C. Sur la courbe, j'ai opté pour une valeur fictive de 10K.

La résistance PTC

La résistance de la thermistance PTC varie inversement à celle d'une thermistance NTC. PTC signifie *Positive Temperature Coefficient*. La conductance d'une PTC diminue (et la résistance augmente) en fonction de la température. Le symbole est le suivant :

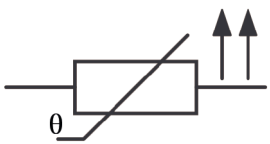
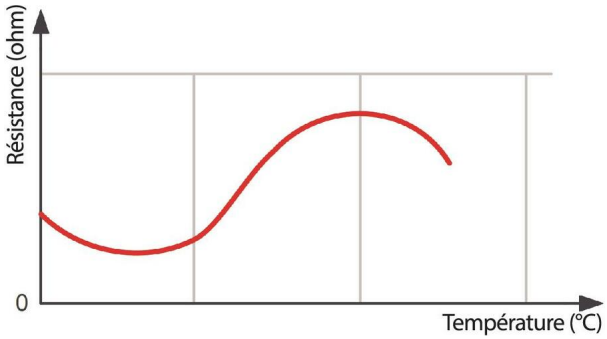


Figure 13-5
Symbole de la PTC (thermistance PTC)

Le sens des flèches permet là aussi d'identifier le type de thermistance. Une hausse de la température entraîne une hausse de la résistance. La courbe caractéristique d'une PTC est l'inverse de celle de la NTC, avec quelques

particularités supplémentaires. En effet, elle peut avoir un minimum dans la région des basses températures et un maximum dans celle des températures élevées.

Figure 13-6 ▶
Courbe caractéristique d'une PTC



Le tableau suivant résume le comportement des deux types de résistances sensibles à la température (NTC et PTC).

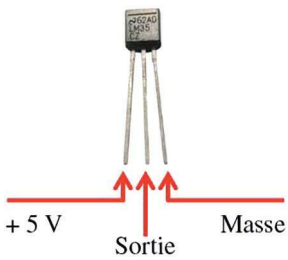
Tableau 13-1 ▶
Comportement des NTC et PTC selon la tempréature

Type	Température	Résistance	Courant
NTC	↑	↓	↑
	↓	↑	↓
PTC	↑	↑	↓
	↓	↓	↑

Le capteur de précision LM35

Nous avons vu que les courbes caractéristiques des deux thermistances n'étaient pas linéaires. Aussi voudrais-je vous présenter un capteur de température qui fait fort bien les choses en produisant une tension de sortie linéaire proportionnelle à la température. Il a pour nom LM35 et présente trois pattes de raccordement. Deux d'entre elles servent à l'alimentation, la troisième de sortie.

Figure 13-7 ▶
Capteur de température LM35 en boîtier plastique TO-92, avec son brochage



On pourrait croire que ce composant s'apparente à un transistor ordinaire mais les apparences sont trompeuses. Ce n'est pas parce qu'il ressemble à un transistor que c'en est un. En cas de doute, consultez sa dénomination qui indique bien qu'il s'agit d'un capteur de température.

Ce capteur convertit la température mesurée en une valeur de tension analogique qui est proportionnelle à la température. Cela s'appelle un *comportement de tension proportionnel à la température*. Le capteur a une sensibilité de 10 mV/°C et une gamme de température comprise entre 0 et 100 °C. La formule pour calculer la température en fonction de la valeur mesurée à l'entrée analogique est la suivante :

$$Température\,[°C] = \frac{5.0 \cdot 100.0 \cdot analogValue}{1024.0}$$

Les valeurs de la formule se justifient ainsi :

- 5.0 : tension de référence Arduino de 5 V ;
- 100.0 : valeur maximale mesurable par le capteur de température ;
- 1024 : résolution de l'entrée analogique.

Avant d'envoyer la valeur mesurée à l'afficheur sept segments, il paraît intéressant de représenter graphiquement l'évolution de la courbe de température dans Processing. Voilà ce que cela donne :

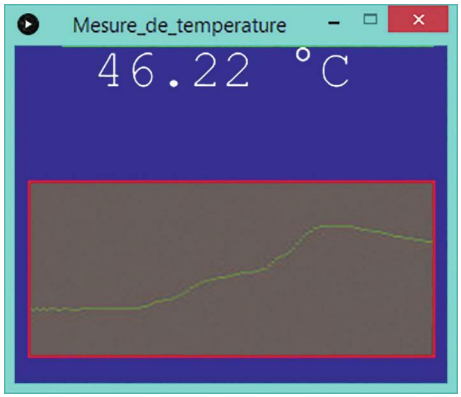


Figure 13-8
Courbe de température dans Processing

La température s'affiche sous la forme d'une valeur de température et sous celle d'une courbe graphique en fonction du temps.

Composants nécessaires

Ce montage nécessite les composants suivants.

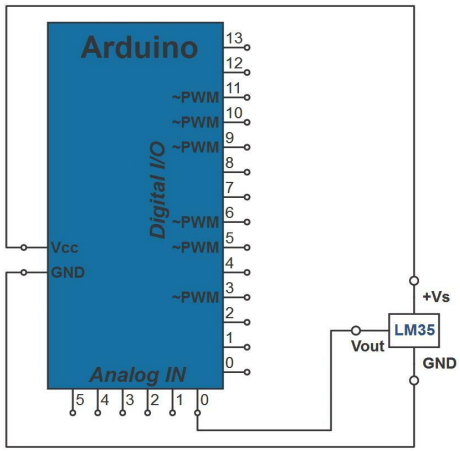
Tableau 13-2 ▶
Liste des composants

Composant	
1 capteur de température LM35	
2 registres à décalage 74HC595	
2 afficheurs sept segments (par exemple de type SA 39-11 GE avec anode commune)	
14 résistances de 330 Ω	
1 résistance de 10K	
1 bouton-poussoir miniature	

Schéma

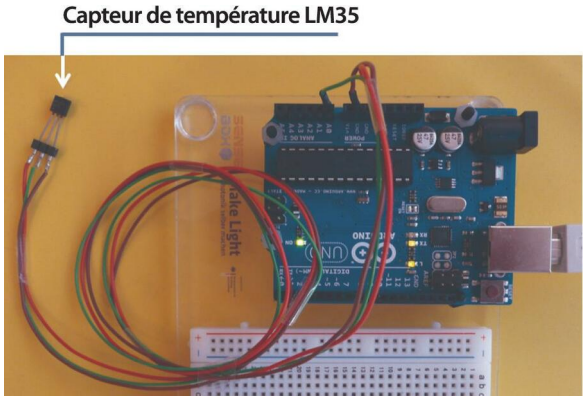
Voici le schéma d'interrogation de la valeur de température à l'aide de la thermistance LM35 :

Figure 13-9 ▶
Schéma de mesure de la température



Réalisation du circuit

Le circuit contient le capteur de température LM35 qui a été raccordé à un petit connecteur tripolaire afin de pouvoir raccorder facilement un câble de rallonge.



◀ Figure 13-10
Réalisation du circuit de mesure de la température avec le LM35

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

Le sketch suivant lit la valeur sur l'entrée analogique A0 et stocke plusieurs valeurs mesurées dont la valeur moyenne est ensuite transmise à l'interface série.

```
#define LM35  A0        // Connexion à la sortie du LM35
#define DELAY 10        // Valeur de la pause
const int cycles = 20;  // Nombre de mesures

void setup() {
  Serial.begin(9600);
}

void loop() {
  float resultTemp = 0.0;
  for(int i = 0; i < cycles; i++) {
    int analogValue = analogRead(LM35);
    float temperature = (5.0 * 100.0 * analogValue) / 1024;
    resultTemp += temperature; // Addition des valeurs mesurées
    delay(DELAY);              // Courte pause
  }
  resultTemp /= cycles;        // Calcul de la moyenne
  Serial.println(resultTemp); // Envoi à l'interface série
}
```

Examinons la signification de ce sketch.

Revue de code

La valeur déterminée par le capteur de température LM35 est calculée avec la formule ci-après :

```
float temperature = (5.0 * 100.0 * analogValue) / 1024;
```

et moyennée à l'aide d'une boucle `for`. Les valeurs mesurées y sont additionnées, puis la moyenne est calculée. Cette dernière est enfin transmise à l'interface série :

```
Serial.println(resultTemp);
```

Son traitement par Processing commence immédiatement.

Revue de code Processing

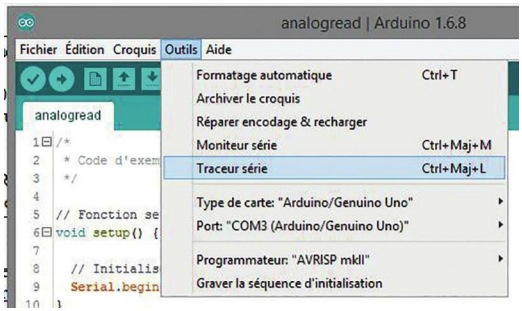
```
import processing.serial.*;
Serial mySerialPort;
float realTemperature;
int temperature, xPos;
int[] yPos;
PFont font;

void setup() {
  size(321, 250); smooth();
  println(Serial.list());
  mySerialPort = new Serial(this, Serial.list()[0], 9600);
  mySerialPort.bufferUntil('\n');
  yPos = new int[width];
  for(int i = 0; i < width; i++)
    yPos[i] = 250;
  font = createFont("Courier New", 40, false);
  textFont(font, 40); textAlign(RIGHT);
}

void draw() {
  background(0, 0, 255, 100);
  strokeWeight(2); stroke(255, 0, 0);
  fill(100, 100, 100); rect(10, 100, width - 20, 130);
  strokeWeight(1); stroke(0, 255, 0);
  int yPosPrev = 0, xPosPrev = 0;
  // Décaler les valeurs du tableau vers la gauche
  for(int x = 1; x < width; x++)
    yPos[x-1] = yPos[x];
  // Ajout des nouvelles coordonnées de la souris
  // à l'extrémité droite du tableau
```

```
  yPos[width-1] = temperature;
  // Affichage du tableau
  for(int x = 10; x < width - 10 ; x++)
    point(x, yPos[x]);
  fill(255);
  text(realTemperature + " °C", 250, 30); // Celsius
  delay(100);
}

void serialEvent(Serial mySerialPort) {
  String portStream = mySerialPort.readString();
  float data = float(portStream);
  realTemperature = data;
  temperature = height - (int)map(data, 0, 100, 0, 130) - 25;
  println(realTemperature);
}
```

Nous en arrivons bientôt à l'objectif de ce montage qui est la réalisation d'une mesure de la température et l'affichage de la valeur sur les deux afficheurs à sept segments. Mais avant cela, j'aimerais revenir sur une possibilité intéressante de l'IDE Arduino.

Affichage de valeurs analogiques sur l'IDE Arduino

Vous avez vu comment représenter l'évolution des valeurs mesurées à l'aide de Processing, ce qui permet de rendre la variation des températures plus lisible. Toutefois, l'IDE Arduino dispose d'une fonctionnalité qui permet de présenter très facilement cette évolution. Il s'agit du *Serial Plotter* ou *Traceur série*. Chargez le premier sketch de ce montage sur votre carte Arduino, puis sélectionnez la commande *Outils* | *Traceur série*.



La fenêtre qui apparaît représente l'évolution de la valeur de température dans le temps.

Figure 13-11 ▶
Traceur série

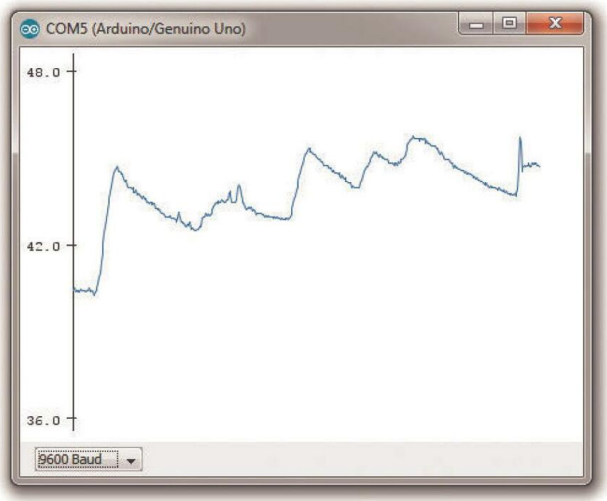


Schéma élargi

Voici le schéma pour l'interrogation de la valeur de température à l'aide de la thermistance LM35 et l'affichage sur les afficheurs sept segments :

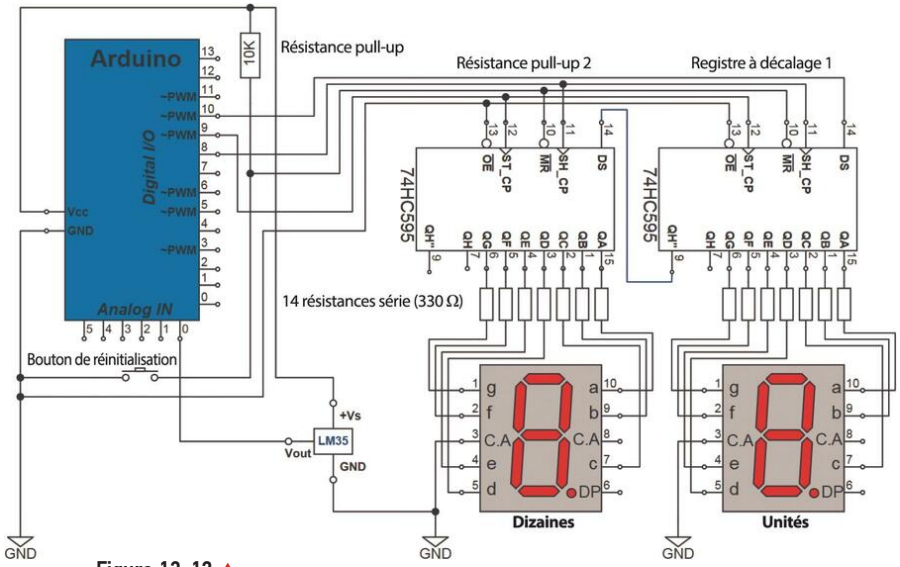
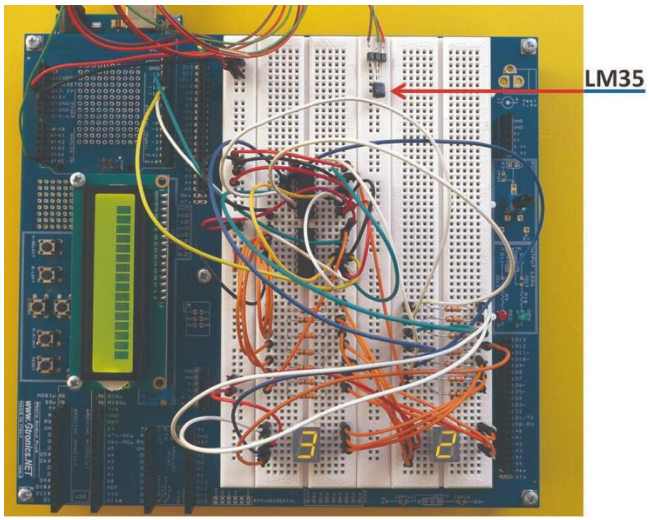


Figure 13-12 ▲
Schéma de mesure de la température et d'affichage sur les afficheurs sept segments

Réalisation du circuit étendu

Le circuit contient le capteur de température LM35 qui a été raccordé à deux afficheurs sept segments.



◀ Figure 13-13
Réalisation du circuit de mesure de la température avec le LM35

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

Le sketch suivant lit la valeur sur l'entrée analogique A0 et stocke plusieurs valeurs mesurées dont la valeur moyenne est ensuite transmise à l'interface série.

```
#define LM35 A0                  // Broche LM35
byte segments[10]= {B01000000,  // 0
                    B01111001,  // 1
                    B00100100,  // 2
                    B00110000,  // 3
                    B00011001,  // 4
                    B00010010,  // 5
                    B00000010,  // 6
                    B01111000,  // 7
                    B00000000,  // 8
                    B00010000}; // 9
int shiftPin = 8;                // SH_CP
int storagePin = 9;              // ST_CP
int dataPin = 10;                // DS
```

```
// Fonction de transmission des informations
void sendBytes(int value) {
    digitalWrite(storagePin, LOW);
    shiftOut(dataPin, shiftPin, MSBFIRST, value >> 8);
    shiftOut(dataPin, shiftPin, MSBFIRST, value & 255);
    digitalWrite(storagePin, HIGH);
}

void displayValue(int value) {
  byte tens = int(value / 10);    // Calculer le chiffre des dizaines
  byte units = value - tens * 10; // Calculer le chiffre des unités
  sendBytes(segments[tens] << 8 | segments[units]);  // Dizaines et unités
}

void setup() {
  Serial.begin(9600);
  pinMode(shiftPin, OUTPUT);
  pinMode(storagePin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  int analogValue = analogRead(LM35);
  int temperature = (5.0 * 100.0 * analogValue) / 1024;
  displayValue(temperature);   // Afficher la valeur
  Serial.println(temperature); // Envoi à l'interface série
  delay(500);                  // Courte pause
}
```

Examinons la signification de ce sketch.

Revue de code

Je n'ai pas grand-chose à ajouter, car vous connaissez déjà les bases. Comme les afficheurs sept segments affichent uniquement des nombres entiers, nous avons défini la valeur de température comme étant de type `int`, ce qui signifie qu'un nombre décimal doit être converti en nombre entier. J'ai choisi pour simplifier de ne pas calculer la moyenne des températures.

Peut-être avez vous remarqué que les modes d'écriture varient considérablement sur Internet et dans les exemples de sketches d'interrogation d'une entrée analogique. À certains endroits, l'entrée analogique `0` sera uniquement désignée par le chiffre `0` et à d'autres par `A0`. Quelle est la différence ?

En principe, il n'y a pas de différence. La forme d'écriture `A0`, qui est un *alias*, a été permise, car elle est plus claire puisque l'on sait immédiatement que l'on a affaire à une broche analogique.

Vous trouverez de plus amples informations à l'adresse suivante :

https://www.arduino.cc/en/Tutorial/AnalogInputPins

Problèmes courants

- Si l'afficheur sept segments affiche des chiffres qui ne paraissent pas cohérents, vérifiez les valeurs transmises dans le moniteur série.

Qu'avez-vous appris ?

- Vous avez vu la différence et les spécificités des thermistances NTC et PTC.
- Vous avez utilisé un capteur LM35 afin de mesurer la température.
- Nous avons d'abord affiché les valeurs dans Processing, puis à l'aide de deux afficheurs sept segments.
- Vous avez représenté une courbe des valeurs de température mesurées à l'aide du Traceur série de l'IDE Arduino de la même façon que dans Processing.

Montage

14 Le clavier numérique

Jusqu'à présent, comme entrée numérique, nous avons utilisé un ou plusieurs boutons-poussoirs miniatures qui étaient disposés de manière assez arbitraire sur la plaque d'essais. Dans ce montage, nous allons réunir plusieurs boutons sur un shield Arduino afin de fabriquer un dispositif d'entrée confortable. Nous en profiterons également pour voir comment fabriquer son propre shield.

Qu'est-ce qu'un clavier numérique ?

Vous connaissez déjà le bouton-poussoir, dont nous avons parlé au cours de certains montages. Mais ici, plusieurs boutons-poussoirs (touches) sont réunis en une matrice – donc disposés en lignes et colonnes – de manière à proposer les chiffres 0 à 9 et deux touches spéciales telles que * et #. À quoi cela sert-il ? Eh bien, vous utilisez ce jeu de touches tous les jours, entre autres pour téléphoner.

◀ **Figure 14-1**
Clavier de téléphone

Il s'agit d'une matrice de 4 × 3 touches (4 lignes et 3 colonnes). Cette matrice est également appelée *keypad* (clavier numérique) et peut être achetée prête à l'emploi en différentes variantes. La figure 14-2 montre deux *claviers numériques à film*. Celui de gauche possède même quelques touches supplémentaires A à D, qui peuvent s'avérer très utiles si les 12 touches du clavier numérique de droite ne suffisent pas pour votre projet.

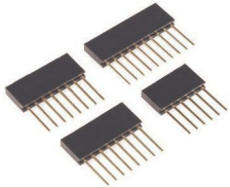
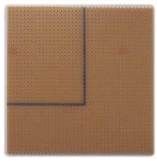
Figure 14-2 ▶
Clavier numérique à film 4 × 4 à 16 touches et clavier numérique à film 4 × 3 à 12 touches

Mais comment brancher par exemple le clavier numérique à film 4 × 4 sur votre Arduino sans rencontrer des problèmes de broches ? On pourrait bien sûr raccorder les 16 touches d'un côté au +5 V et les 16 prises correspondantes aux entrées numériques.

Vous pourriez bien sûr procéder ainsi et cela fonctionnerait s'il n'y avait pas les limites physiques de la carte Arduino Uno. Une solution consisterait à utiliser la carte Arduino Mega, dont le nombre d'interfaces est bien élevé. Mais soyons ingénieux ; il existe une bibliothèque pour claviers numériques prête à l'emploi sur le site Internet Arduino, aussi allons-nous tout faire par nous-mêmes. Nous utiliserons le clavier numérique 4 × 3 que nous aurons fabriqué de nos propres mains. Voici la liste du matériel nécessaire.

Composants nécessaires

Ce montage nécessite les composants suivants.

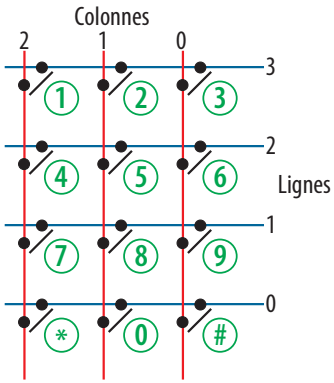
Composant	
12 boutons-poussoirs miniatures	
1 jeu de connecteurs femelles empilables • 2 connecteurs à 8 broches • 1 connecteur à 6 broches • 1 connecteur à 10 broches	
1 carte de dimensions 10 × 10 ou mieux 16 × 10 (vous pourrez alors en faire deux shields). La découpe du shield est déjà indiquée, et je reviendrai bientôt sur cette dernière.	

◀ Tableau 14-1
Liste des composants

Réflexions préliminaires

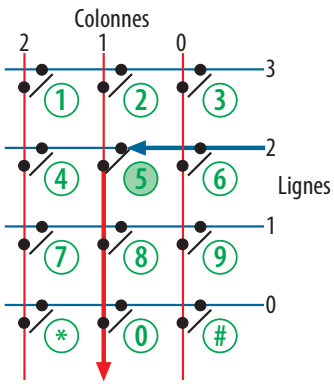
Nous venons de voir que notre clavier numérique 4 × 4 nécessitait 16 lignes pour interroger toutes les touches. Un clavier numérique 4 × 3 n'aurait quant à lui besoin que de 12 lignes. Mais ce serait encore beaucoup trop à mon avis. Une solution astucieuse existe, dont l'idée de base a déjà servi pour commander les deux afficheurs sept segments. Vous vous demandez sûrement ce que des afficheurs sept segments ont à voir avec ces touches. Le mot commun est *multiplexage*. Il signifie que certains signaux sont regroupés et envoyés par un moyen de transmission pour minimiser l'utilisation des lignes et en tirer profit le plus possible. Sur les afficheurs sept segments, les lignes de commande de deux segments sont montées en parallèle et utilisées pour commander les deux. Sept ou huit lignes par segment sont ainsi économisées. La solution trouvée pour interroger les différentes touches d'un clavier numérique est relativement simple. Voici le câblage des 12 touches.

Figure 14-3 ▶
Câblage des 12 touches d'un clavier numérique 4 × 3



Imaginez une grille composée de 4 × 3 fils métalliques posés l'un sur l'autre sans pour autant se toucher. Voilà à quoi ressemble ce graphique. On voit que les 4 fils horizontaux, en bleu, forment des lignes numérotées de 0 à 3. Au-dessus, les trois fils verticaux en rouge forment à peu de distance des colonnes numérotées de 0 à 2. Chaque intersection présente des petits contacts reliant, quand on appuie sur la touche, la ligne et la colonne en question pour former un tronçon de circuit électrique. Regardez bien la figure 14-4, où la touche numéro 5 est enfoncée.

Figure 14-4 ▶
La touche 5 est enfoncée (les lignes en gras montrent le passage du courant).



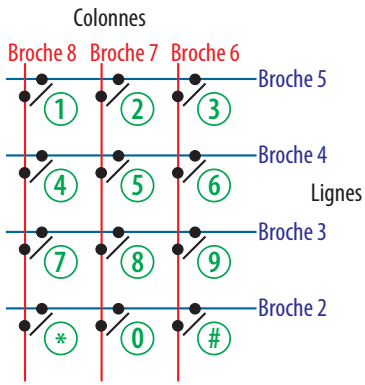
Le courant peut alors passer de la ligne 2 via l'intersection numéro 5 dans la colonne 1 et y être détecté. Plus tard, nous utiliserons les résistances pull-up internes du microcontrôleur, ce qui nous épargnera un raccordement externe avec des résistances séparées. La commande se fait aussi avec un niveau `LOW`. Nous y reviendrons.

Mais si une tension est appliquée simultanément à toutes les lignes, la touche 2 au-dessus peut tout aussi bien être pressée et nous enregistrerions une impulsion correspondante sur la colonne 1. Comment faire la différence ?

Disons grossièrement que nous envoyons tour à tour un signal par les lignes 0 à 3 et interrogeons ensuite également tour à tour le niveau sur les colonnes 0 à 2. Le déroulement est alors le suivant, la commande se faisant avec un niveau `LOW`, comme je l'ai déjà mentionné :

- Niveau `LOW` sur le fil de la rangée 0
  - Interrogation du niveau sur la colonne 0
  - Interrogation du niveau sur la colonne 1
  - Interrogation du niveau sur la colonne 2
  - ...
- Niveau `LOW` sur le fil de la rangée 1
  - Interrogation du niveau sur la colonne 0
  - Interrogation du niveau sur la colonne 1
  - Interrogation du niveau sur la colonne 2
  - ...

Cette interrogation est bien sûr si rapide que suffisamment de passages ont lieu en une seule seconde pour que pas un appui de touche ne soit omis. Le shield a été câblé à demeure avec les numéros de broches des entrées et sorties indiqués sur la figure 14-5.



◀ Figure 14-5
Câblage des différentes lignes et colonnes avec les broches numériques

Pimentons ici un peu les choses et créons notre propre bibliothèque qui servira plus tard à d'autres montages. Elle offre une certaine fonctionnalité de base et pourra bien entendu être modifiée ou élargie si besoin est. Le sketch principal demande continuellement au shield quelle touche a été pressée. Le résultat est affiché dans le moniteur série pour visualisation. Pour vérifier le bon fonctionnement de la bibliothèque, fixons-nous les spécifications suivantes :

- si vous n'appuyez sur aucune touche, aucun caractère n'est affiché dans le moniteur série ;
- si vous n'appuyez que brièvement sur une touche, le chiffre ou le caractère s'affiche dans le moniteur ;

si vous appuyez sur une touche un long moment, qui peut être préalablement défini en conséquence, le chiffre ou le caractère s'affiche plusieurs fois l'un derrière l'autre jusqu'à ce que la touche soit relâchée. Examinons le schéma avant de nous lancer dans la programmation.

Schéma

Nous utilisons uniquement les sept connexions du clavier numérique vers la carte Arduino.

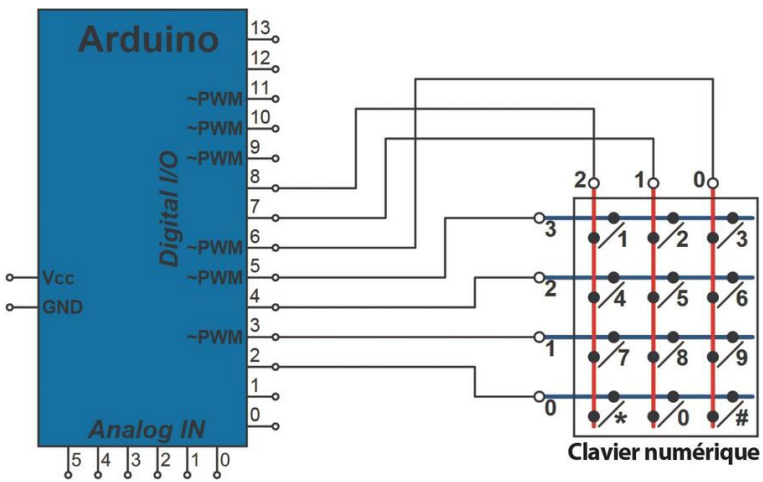
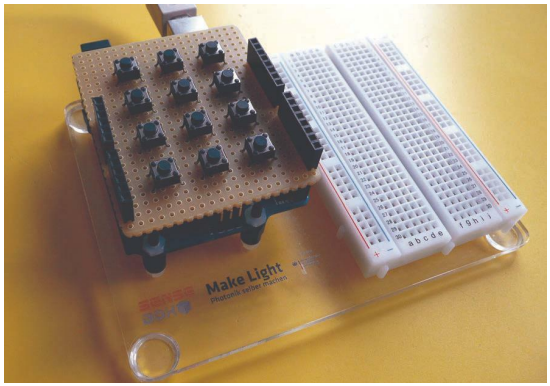


Figure 14-6 ▶
Circuit de commande du clavier numérique

Réalisation du circuit

Voici comment le shield du clavier numérique est branché sur la carte Arduino :



◀ **Figure 14-7**
Réalisation du circuit de commande du clavier numérique

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

La programmation se divise en un sketch principal et une bibliothèque.

Sketch principal avec revue de code

```
#include "MyKeyPad.h"
int rowArray[] = {2, 3, 4, 5};    // Initialiser le tableau avec les numéros
                                  // de broche des lignes
int colArray[] = {6, 7, 8};       // Initialiser le tableau avec les
                                  // numéros de broche des colonnes
MyKeyPad myOwnKeyPad(rowArray, colArray); // Instanciation d'un objet

void setup() {
  Serial.begin(9600);               // Préparer la sortie série
  myOwnKeyPad.setDebounceTime(500); // Régler le temps du rebond à 500 ms
}

void loop() {
  char myKey = myOwnKeyPad.readKey(); // Lecture de la touche pressée
  if(myKey != KEY_NOT_PRESSED)        // Une touche quelconque a-t-elle
                                      // été pressée ?
    Serial.println(myKey);            // Affichage du caractère de la
                                      // touche
}
```

Tout comme vous l'avez appris dans le montage n° 8 « Comment créer une bibliothèque ? », la première ligne permet d'inclure le fichier d'en tête permettant d'utiliser la bibliothèque. Nous verrons bientôt ce qu'il contient. Déclarons pour commencer deux tableaux, que nous initialisons avec les numéros des broches de connexion aux lignes et aux colonnes du clavier numérique. Cela offre une plus grande flexibilité et permet d'adapter les réalisations différentes. La ligne

```
MyKeyPad myOwnKeyPad(rowArray, colArray);
```

génère l'instance `myOwnKeyPad` de la classe `MyKeyPad` qui est définie dans la bibliothèque, et transmet les deux tableaux au constructeur de la classe. Ces informations lui sont nécessaires pour commencer à évaluer sur laquelle des 12 touches on a appuyé. Le temps de rebond est déterminé par la ligne suivante :

```
myOwnKeyPad.setDebounceTime(500);
```

La méthode `setDebounceTime` avec l'argument `500` est ainsi appelée. L'instance est ensuite continuellement interrogée au sein de la fonction `loop`, la question posée étant : « Indique-moi la touche qui est actuellement pressée sur le clavier numérique ! » Pour y arriver, il faut écrire la ligne suivante :

```
char myKey = myOwnKeyPad.readKey();
```

Elle affecte le résultat de la requête à la variable `myKey` du type `char`. On peut maintenant réagir en conséquence. Il le faut, car la méthode renvoie toujours une valeur, qu'une touche ait été pressée ou non. Mais vous souhaitez sûrement voir à l'écran si une touche a été enfoncée. Aussi la valeur `KEY_NOT_PRESSED` est-elle renvoyée quand aucune touche n'est pressée.

```
if(myKey != KEY_NOT_PRESSED)
   Serial.println(myKey);
```

n'envoie donc le caractère correspondant à la touche au moniteur série que si une touche est véritablement appuyée.

Il se peut que vous vous demandiez ce qu'il y a derrière `KEY_NOT_PRESSED`. Question pertinente, car j'en serais venu de toute façon à parler du fichier d'en-tête. De nombreuses constantes symboliques y sont définies. Parmi ces constantes se cache le caractère `-`, qui est toujours envoyé lorsqu'aucune touche n'est pressée. Je lui ai donné ce nom évocateur pour que le code soit plus lisible.

Fichier d'en-tête avec revue de code

Le fichier d'en-tête sert, comme nous l'avons déjà expliqué, à faire connaître les variables et les méthodes nécessaires à la définition de la classe en question. Voyons maintenant ce qu'on y trouve :

```
#ifndef MYKEYPAD_H
#define MYKEYPAD_H
#include <Arduino.h>
#define KEY_NOT_PRESSED '-' // Nécessaire si aucune touche n'est pressée
#define KEY_1 '1'
#define KEY_2 '2'
#define KEY_3 '3'
#define KEY_4 '4'
#define KEY_5 '5'
#define KEY_6 '6'
#define KEY_7 '7'
#define KEY_8 '8'
#define KEY_9 '9'
#define KEY_0 '0'
#define KEY_STAR '*'
#define KEY_HASH '#'

class MyKeyPad {
  public:
    MyKeyPad(int rowArray[], int colArray[]); // Constructeur paramétré
    void setDebounceTime(unsigned int debounceTime); // Réglage du temps
                                                     // de rebond
    char readKey(); // Détermine la touche
                    // pressée sur le clavier numérique
  private:
    unsigned int debounceTime; // Variable locale pour temps de rebond
    long lastValue;            // Dernière valeur de la fonction millis
    int row[4];                // Tableau pour les lignes
    int col[3];                // Tableau pour les colonnes
};
#endif
```

La partie supérieure est consacrée aux constantes symboliques et aux caractères correspondants. Vient ensuite la définition formelle de la classe sans formulation du code qui, comme chacun sait, se trouve dans le fichier `.cpp`.

Mais que veut dire `unsigned int` dans la déclaration des variables ? Le type de donnée `int` vous est déjà familier. Son domaine s'étend des valeurs négatives aux valeurs positives. Le mot-clé `unsigned` placé devant indique que la variable est déclarée sans signe, autrement dit son domaine de valeurs double puisque les valeurs négatives sont supprimées. Ce type de donnée nécessite également (comme `int`) deux octets pour que les valeurs positives soient toutes représentées. Le domaine de valeurs va de 0 à 65 535.

Fichier ccp avec revue de code

Voici maintenant un peu de code « fait maison » :

```
#include "MyKeyPad.h"
// Constructeur paramétré
MyKeyPad::MyKeyPad(int rowArray[], int colArray[]) {
  // Copier le tableau des broches
  for(int r = 0; r < 4; r++)
    row[r] = rowArray[r];
  for(int c = 0; c < 3; c++)
    col[c] = colArray[c];
  // Programmation des broches numériques
  for(int r = 0; r < 4; r++)
    pinMode(row[r], OUTPUT);
  for(int c = 0; c < 3; c++)
    pinMode(col[c], INPUT_PULLUP);
  // Définition initiale de debounceTime à 300 ms
  debounceTime = 300;
}

// Méthode pour régler le temps de rebond
void MyKeyPad::setDebounceTime(unsigned int time) {
debounceTime = time;
}

// Méthode pour déterminer la touche appuyée sur le clavier numérique
char MyKeyPad::readKey() {
  char key = KEY_NOT_PRESSED;
  for(int r = 0; r < 4; r++) {
    digitalWrite(row[r], LOW);
    for(int c = 0; c < 3; c++) {
      if((digitalRead(col[c]) == LOW)&&(millis() - lastValue) >=
                                    ➥ debounceTime) {
        if((c==2)&&(r==3)) key = KEY_1;
        if((c==1)&&(r==3)) key = KEY_2;
        if((c==0)&&(r==3)) key = KEY_3;
        if((c==2)&&(r==2)) key = KEY_4;
        if((c==1)&&(r==2)) key = KEY_5;
        if((c==0)&&(r==2)) key = KEY_6;
        if((c==2)&&(r==1)) key = KEY_7;
        if((c==1)&&(r==1)) key = KEY_8;
        if((c==0)&&(r==1)) key = KEY_9;
        if((c==2)&&(r==0)) key = KEY_STAR; // *
        if((c==1)&&(r==0)) key = KEY_0;
        if((c==0)&&(r==0)) key = KEY_HASH; // #
        lastValue = millis();
      }
    }
  digitalWrite(row[r], HIGH); // Restauration du niveau initial
  }
  return key;
}
```

Voyons d'abord le constructeur. Il sert à initialiser l'objet à créer et à lui donner des valeurs initiales définies. Un constructeur doit permettre d'initialiser, autant que possible complètement, l'instance, de telle sorte qu'aucun appel de méthode ne soit plus en principe nécessaire pour l'initialisation. Elles ne sont plus utilisées que pour corriger certains paramètres qui, le cas échéant, doivent, ou peuvent, être modifiés en cours de sketch. Le constructeur n'est appelé *qu'une seule fois* et *de manière implicite* lors de l'instanciation, et après cela plus jamais dans la vie de l'objet. Dans notre exemple, les tableaux des lignes et des colonnes lui sont communiqués lors de l'appel, de manière à pouvoir être transmis ensuite aux tableaux locaux au moyen de deux boucles `for` :

```
// Copie des tableaux de broches
for(int r = 0; r < 4; r++)
  row[r] = rowArray[r];
for(int c = 0; c < 3; c++)
  col[c] = colArray[c];
```

Les broches numériques sont ensuite initialisées et leurs sens de transfert sont définis :

```
// Programmation des broches numériques
for(int r = 0; r < 4; r++)
  pinMode(row[r], OUTPUT);
for(int c = 0; c < 3; c++)
  pinMode(col[c], INPUT_PULLUP);
// Définition initiale de debounceTime à 300 ms
debounceTime = 300;
```

Nous avons vu qu'un objet devait toujours être complètement instancié au moyen d'un constructeur. Mais ici nous lui transmettons uniquement les tableaux de broches pour les lignes et les colonnes. Un autre paramètre important est néanmoins le temps de rebond. Celui-ci n'est pourtant pas transmis à l'objet par le constructeur. Nous avons pour ce faire une méthode propre. Cela ne contredit-il pas ce qui vient d'être dit ?

Oui et non ! Il est vrai que le constructeur ne connaît pas le temps de rebond. Mais regardez sa dernière ligne. Le temps y est réglé sur 300 ms. Il s'agit pratiquement d'une initialisation *câblée en dur*, comme on dit si bien dans les milieux de la programmation. Si la valeur ne vous dit rien, vous pouvez toujours l'adapter à vos besoins, tout comme je l'ai fait d'ailleurs pour la méthode `setDebounceTime`. La valeur de `300` (ms) m'a semblé ici convenir. J'aurais évidemment pu la définir directement, mais je voulais vous montrer cette possibilité. La tâche en question est accomplie par la méthode `readKey`, qui est appelée sans cesse dans la boucle `loop` pour pouvoir réagir immédiatement à un appui sur une touche. Au début de

l'appel de la méthode, la ligne suivante fait en sorte que la variable key soit immédiatement pourvue d'une valeur initiale :

```
char key = KEY_NOT_PRESSED;
```

Allez voir dans le fichier d'en-tête de quelle valeur il s'agit. Si aucune touche n'est en effet pressée, c'est précisément ce signe qui est réexpédié comme résultat. Vient ensuite l'appel des deux boucles `for` imbriquées l'une dans l'autre. La première ligne du clavier numérique est mise au niveau `LOW` par :

```
digitalWrite(row[r], LOW);
```

Les niveaux de toutes les colonnes sont ensuite testés.

```
...
for(int r = 0; r < 4; r++) {
   digitalWrite(row[r], LOW);
   for(int c = 0; c < 3; c++) {
     if((digitalRead(col[c]) == LOW)&&(millis() - lastValue) >=
                                    ➥ debounceTime) {
       if((c==2)&&(r==3)) key = KEY_1;
       if((c==1)&&(r==3)) key = KEY_2;
       if((c==0)&&(r==3)) key = KEY_3;
       if((c==2)&&(r==2)) key = KEY_4;
       if((c==1)&&(r==2)) key = KEY_5;
       if((c==0)&&(r==2)) key = KEY_6;
       if((c==2)&&(r==1)) key = KEY_7;
       if((c==1)&&(r==1)) key = KEY_8;
       if((c==0)&&(r==1)) key = KEY_9;
       if((c==2)&&(r==0)) key = KEY_STAR; // *
       if((c==1)&&(r==0)) key = KEY_0;
       if((c==0)&&(r==0)) key = KEY_HASH; // #
       lastValue = millis();
     }
   }
...
```

Si une colonne présente également un niveau `LOW` et si en plus le temps de rebond a été pris en compte, alors la première condition `if` est remplie et toutes les conditions `if` subséquentes sont évaluées. Si une condition est vérifiée pour le compteur de lignes `r` et le compteur de colonnes `c`, la variable key est initialisée avec la valeur initiale correspondante et renvoyée à l'appelant, en fin de méthode, par l'instruction `return`. Une fois la boucle intérieure terminée, les lignes qui viennent d'être mises au niveau `LOW` doivent être remises dans leur état initial, qui est le niveau `HIGH`.

```
digitalWrite(row[r], HIGH); // Restauration du niveau initial
```

Si l'état `LOW` était conservé, une interrogation ciblée d'une certaine ligne ne serait alors plus possible. Toutes les lignes auraient un niveau `LOW` une fois la boucle extérieure terminée, ce qui mettrait toute la logique d'interrogation sens dessus dessous.

La fonction `millis` renvoie le nombre de millisecondes écoulées depuis le début du sketch. La dernière valeur est pour ainsi dire stockée temporairement dans la variable `lastValue`, une fois la boucle intérieure terminée. Si la boucle est de nouveau appelée, la différence entre la valeur actuelle en millisecondes et la valeur précédente est calculée. Ce n'est que si elle est supérieure au temps de rebond défini que la condition est jugée *vérifiée*. Elle se trouve cependant liée avec l'expression qui la précède par un `&` logique.

```
if((digitalRead(col[c]) == HIGH)&&(millis() - lastValue) >=
                              ➥ debounceTime)...
```

Ce n'est que si les deux conditions délivrent le résultat logique *vrai* à l'instruction `if` que la ligne se poursuit avec l'accolade. Cette structure permet d'obtenir une interruption temporelle, qui se produit aussi toute seule dans certains sketches.

Réalisation du shield

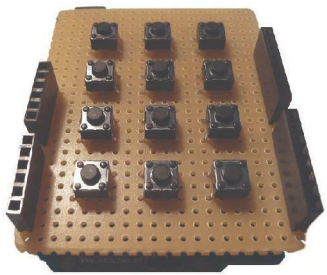
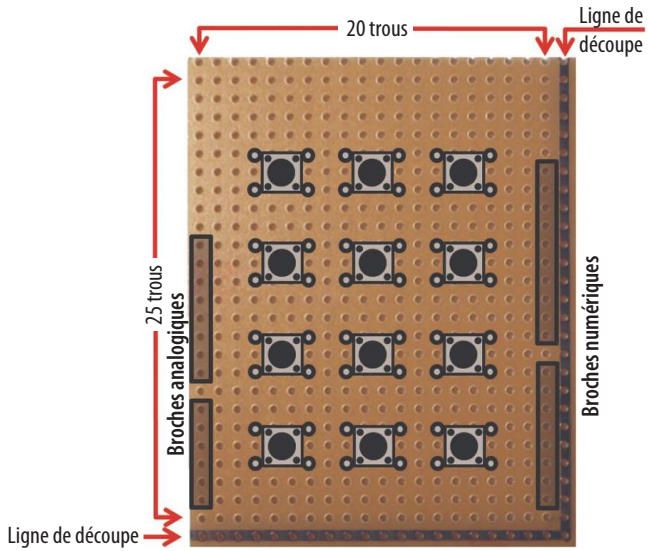


Figure 14-8
Réalisation du clavier numérique avec son propre shield

La construction du shield n'est pas mal du tout, n'est-ce pas ? Je vous avais dit au début que je vous montrerais comment faire une carte à la bonne taille. La carte présentée dans le tableau 14-1 comporte un marquage correspondant à la taille définitive du shield.

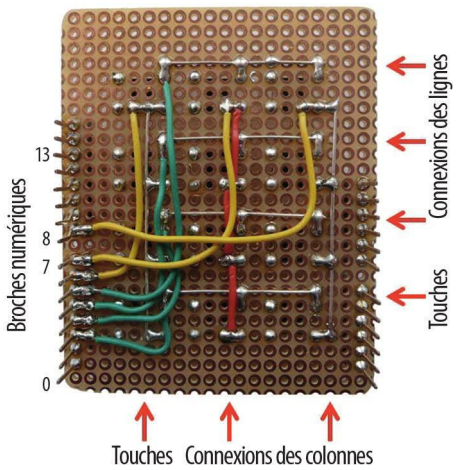
Figure 14-9 ▶
Taille du shield basée sur les écarts entre les trous



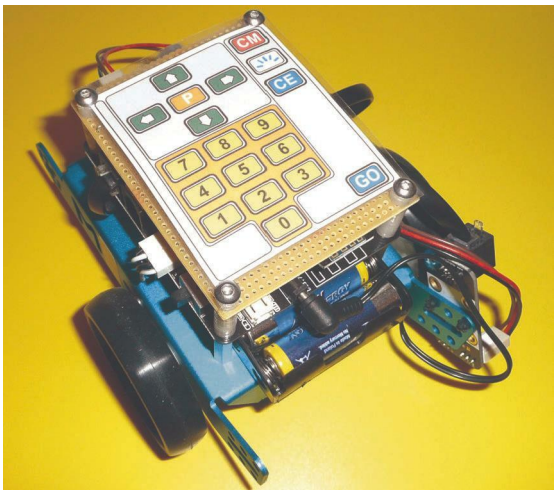
On voit sur l'image les positions exactes des connecteurs femelles et des touches. Il suffit de compter les trous sur la carte et de positionner ensuite les composants. Ne commencez à souder que quand vous avez tout placé sur la carte. Vous évitez ainsi les positionnements incorrects, et les erreurs vous sautent tout de suite aux yeux. Si vous soudez les composants aussitôt après les avoir placés, il se peut que vous vous aperceviez plus tard que vous avez commis une erreur, et que vous ayez tout à dessouder.

La figure 14-10 illustre l'envers de la carte une fois tous les composants soudés et tous les fils raccordés.

Figure 14-10 ▶
Taille du shield basée sur les écarts entre les trous



Les fils verts établissent les liaisons vers les lignes, et les fils jaunes vers les colonnes. Les fils rouges sont les connexions intermédiaires des colonnes, qui passent au-dessus des fils horizontaux. Ce shield permet de fabriquer des claviers spéciaux à partir de modèles adaptés à vos besoins personnels, comme celui illustré sur la figure suivante.



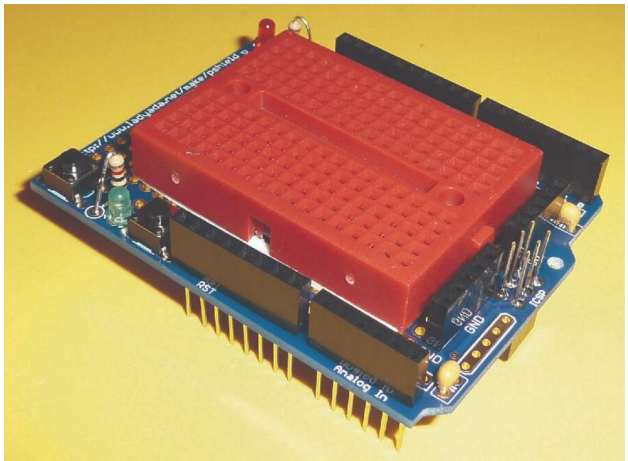
◀ Figure 14-11
Le robot motorisé mBot avec sa commande par pavé numérique

Que diriez-vous de construire un robot motorisé tel que celui-là dans un prochain montage ? Le clavier à film est très facile à fabriquer avec une plastifieuse. Cela présente aussi l'avantage de le protéger contre les doigts gras. Ce clavier numérique fourni avec trois connexions est extrêmement simple d'emploi. Mais, trêve de digressions, pour l'instant, nous allons nous consacrer à la construction d'un shield personnel puisque c'est le sujet de ce montage.

Construction d'un shield Arduino

Le shield Arduino le plus simple disponible dans le commerce est à mon avis le Proto Shield, qui est illustré ci-dessous.

Figure 14-12 ▶
Proto Shield d'Adafruit

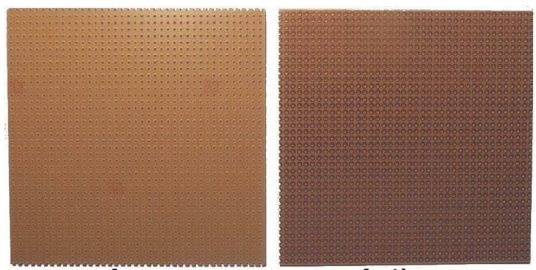


Il y a suffisamment de place sur la face supérieure du Proto Shield pour une petite plaque d'essais afin d'y réaliser de petits circuits. Le shield d'Adafruit (Arduino Kit vR3) illustré n'est pas fourni avec une plaque d'essais, mais vous pouvez y remédier moyennant une somme modique. Voici l'adresse Internet du fabricant du Proto Shield :

https://www.adafruit.com/products/2077

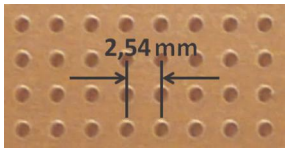
Si je vous dis que la conception de la carte Arduino n'est pas parfaite, vous aurez sans doute du mal à me croire. Pourtant, c'est vrai. Commençons par la carte de circuit imprimé perforée, vendue dans différents formats. La mienne mesure 100 mm × 100 mm et ressemble à celle illustrée ci-dessous.

Figure 14-13
Carte de circuit imprimé perforée



La carte se compose d'un support isolant, par exemple en bakélite ou en résine époxy, et d'une couche de cuivre conductrice. Comme son nom

l'indique, la carte de circuit imprimé perforée présente une multitude de trous régulièrement espacés, bordés d'une couche de cuivre circulaire. Le fil de raccordement d'un composant est enfilé du recto vers le verso de la carte et fixé par soudure à la couche de cuivre.



◀ Figure 14-14
Vue partielle grossie d'une carte de circuit imprimé perforée

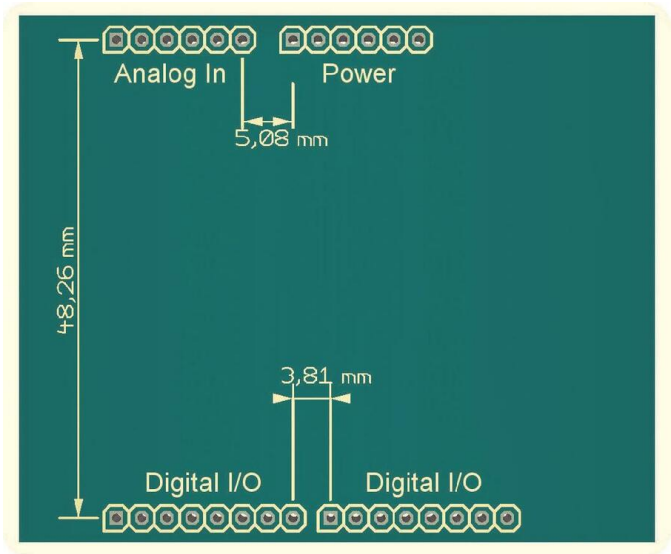
Cette vue grossie montre la distance entre les trous, qui est en règle générale de 2,54 mm. Et c'est là que les choses commencent à se compliquer. Si tout va bien côté carte de circuit imprimé perforée, cette norme n'est en revanche pas respectée du tout côté carte Arduino, et je ne sais pas pourquoi les développeurs l'ont voulue différente.

Les dimensions de la carte de circuit imprimé perforée sont alors les suivantes :

- largeur : 64 mm ;
- hauteur : 53 mm.

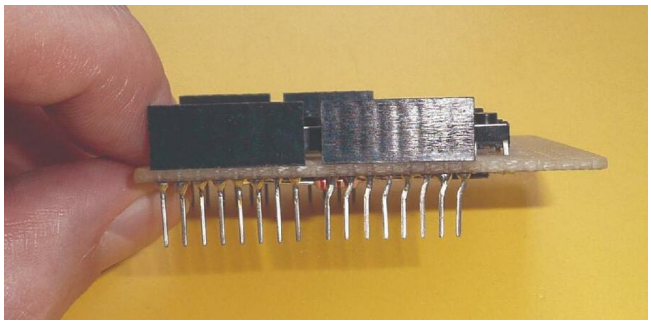
Maintenant un peu de calcul pour comprendre les éloignements des différents trous les uns des autres : les deux rangées du haut pour Analog In et Power, de 6 trous chacune, ne posent aucun problème, car elles sont séparées par un trou libre, autrement dit l'écart est de 2 × 2,54 mm = 5,08 mm. Cela ne pose pas de problème pour la carte de circuit imprimé perforée. Passons aux rangées du bas pour Digital I/O. Pour une raison que j'ignore, l'écart entre ces deux rangées, 3,81 mm environ, n'est même pas un multiple de 2,54 mm, mais est inférieur à deux fois 2,54 mm (soit 5,08 mm). Il n'est donc pas possible en l'état d'utiliser les connecteurs femelles et leurs broches sous cette forme. On voit cependant sur le shield fini que je les ai quand même soudés dans les trous de la carte de circuit imprimé.

Figure 14-15 ▶
Vue de dessus du shield de prototypage

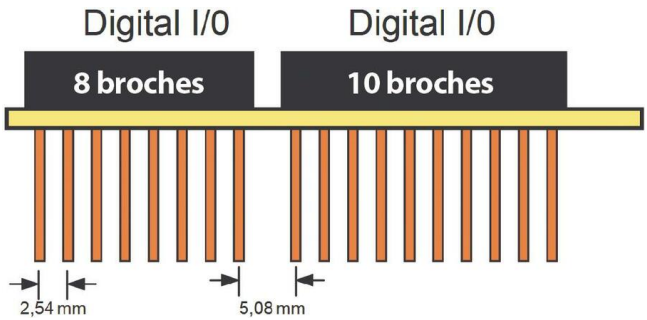


Vous remarquerez que pour adapter le shield fabriqué aux connecteurs de la carte Arduino, il faut tordre les broches ! Ici, il faut déformer un peu les broches du connecteur femelle de droite. On voit, sur la figure suivante, ces broches tordues vers la gauche.

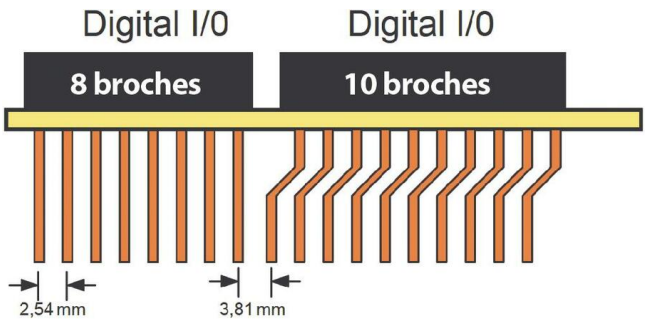
Figure 14-16 ▶
Broches des connecteurs femelles numériques



Les deux figures suivantes « Avant » et « Après » permettent de mieux comprendre ce qu'il faut faire.



◀ Figure 14-17
Avec cet écart de 2 × 2,54 mm = 5,08 mm entre les broches, le shield ne va pas sur la carte Arduino.



◀ Figure 14-18
Les broches ayant été tordues en conséquence, le shield va désormais sur la carte Arduino.

Les broches sont tordues vers la gauche au moyen de la petite pince.

◀ Figure 14-19
Petite pince universelle

Procédez avec soin et ne tordez pas les broches dans tous les sens, car elles peuvent finir pas casser. Ne craignez rien ! Je l'ai fait moi-même et ce n'est pas sorcier. La déformation des broches se fait en deux étapes. On tord d'abord la broche vers la gauche, puis on descend légèrement la pince et on tord à nouveau la broche vers la droite. On obtient ainsi une orientation verticale qui est juste un peu décalée vers la gauche. La broche doit alors se trouver au-dessus d'un trou du connecteur femelle. Procédez de préférence de gauche à droite, en commençant par celle du bout. Lorsque le shield est prêt, vous pouvez souder le bouton-poussoir miniature et les connexions.

Exercice complémentaire

La bibliothèque `KeyPad` fait pour le moment partie du sketch que vous avez créé. Je pense que ce serait une bonne idée de la copier quelque part, à l'attention de tous les autres sketches qui pourraient en avoir besoin. Si vous ne savez plus où, relisez le montage n° 10 du dé électronique, au cours duquel vous aviez créé votre première bibliothèque. Vous y trouverez les informations nécessaires. Il faut pour ce faire rajouter le fichier `keyword.txt` dans votre bibliothèque. Entrez-y les mots-clés nécessaires, qui sont indiqués en couleurs dans l'IDE Arduino.

Problèmes courants

La réalisation de ce shield nécessitant beaucoup de soudure, les erreurs peuvent être d'autant plus nombreuses.

- Vérifiez que les fils sont bien raccordés aux bonnes broches.
- Pas de court-circuit éventuel ? Le mieux est de prendre une loupe et de vérifier chaque soudure. Un court-circuit de la taille d'un cheveu n'est souvent pas visible à l'œil nu.
- Avez-vous correctement relié les différentes touches entre elles, de manière à ce qu'elles forment des lignes et des colonnes ? Servez-vous du schéma.

Qu'avez-vous appris ?

- Vous avez vu qu'on peut fabriquer soi-même un *clavier numérique* avec des composants très simples et peu onéreux. Pour ceux qui ont la patience nécessaire et qui ont envie de faire par eux-mêmes au lieu de toujours se servir des composants tout prêts vendus dans les magasins, ceci a pu être un bon début et leur a permis de montrer ou plutôt d'entretenir leur créativité.
- Le soudage qui, il y a des décennies, était excessivement pratiqué dans les premiers bricolages électroniques, n'est selon moi plus à la mode aujourd'hui. Mais j'espère du moins que l'odeur d'étain fondu et de plastique brûlé vous aura charmé, comme elle a su le faire dans ma jeunesse.
- Nous avons créé ensemble notre propre classe, qui peut servir à interroger la matrice de touches. Vous avez certainement tiré parti des principes de la POO, qui vous avaient été expliqués auparavant.
- Vous avez fabriqué votre propre shield de clavier numérique.

Montage

15

Un afficheur alphanumérique

Pour afficher des informations avec une carte Arduino, on ne peut pas toujours se contenter d'une ou plusieurs LED. Même si cela présente certains avantages, il n'est pas possible de produire des messages nuancés. Nous allons donc nous intéresser à d'autres afficheurs compatibles avec la carte Arduino.

Qu'est-ce qu'un afficheur LCD ?

Que serait un microcontrôleur sans son afficheur pour correspondre avec le monde extérieur sans passer par l'ordinateur ou le moniteur série ? Bien sûr, on a déjà vu comment, par exemple, utiliser des afficheurs sept segments pour représenter des chiffres. S'il s'agit de représenter plusieurs chiffres ou des lettres ou encore des caractères spéciaux tels que *, #, %..., les limites du possible sont vite atteintes. On utilise dans ces cas-là un *afficheur à cristaux liquides* ou *LCD (Liquid Cristal Display)* sous sa forme abrégée. Ces afficheurs contiennent des cristaux liquides capables de modifier leur orientation en fonction d'une tension appliquée, et de jouer ainsi plus ou moins sur l'incidence de la lumière. De tels éléments d'affichage utilisent en général des motifs composés de points *(Dot-Matrix)* pour représenter à peu près tous les signes (chiffres, lettres ou caractères spéciaux). Leur taille et leur équipement varient. Dans l'environnement Arduino, un afficheur LCD avec pilote HD44780 est relativement souvent utilisé. Ce pilote, qui s'est imposé comme le quasi standard, est souvent adapté par beaucoup d'autres fabricants. La figure 15-1 montre un afficheur de ce type.

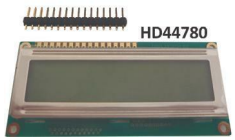
Figure 15-1 ▶
Afficheur LCD

Il existe pour cet élément une bibliothèque livrée avec l'IDE Arduino. Vous pouvez bien entendu raccorder n'importe quel afficheur ou presque, à condition de trouver une bibliothèque appropriée ou de la créer vous-même. Nous utilisons pour notre montage l'afficheur ci-dessus, à 2 lignes de 16 caractères chacune.

Composants nécessaires

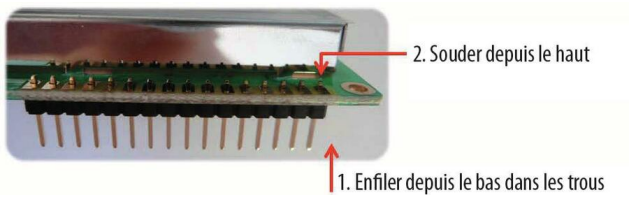
Ce montage nécessite les composants suivants.

Tableau 15-1 ▶
Liste des composants

Composant	
1 LCDisplay HD44780 + barrette à 16 broches	
1 trimmer de 10K ou 20K	

Remarques sur l'utilisation de l'afficheur LCD

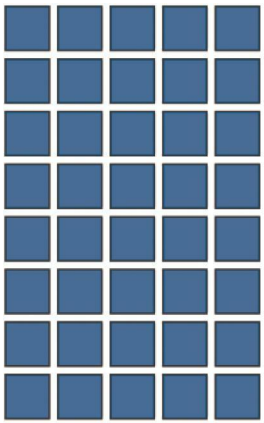
Si vous achetez un afficheur LCD tout neuf, il se peut que seuls des trous de connexion soient présents sur le circuit imprimé, comme vous pouvez le voir sur l'image ci-dessus. Vous pouvez alors, soit équiper de fils les contacts nécessaires et vous en servir plus tard pour le circuit sur la plaque d'essais, soit – et c'est le mieux – vous procurer une barrette à broches, comme vous pouvez le voir également sur l'image en haut de la page suivante. Des barrettes sont par exemple proposées avec une rangée de 40 broches et un pas de 2,54 mm. Coupez-les à une longueur de 15 broches en les pliant délicatement à l'endroit souhaité. Allez-y doucement, car elles ont tendance à casser là où on ne veut pas. Enfilez ensuite les broches de la barrette depuis le bas dans les trous et soudez-les sur la face supérieure.



Vous pouvez ainsi brancher sans problème le module sur votre plaque d'essais.

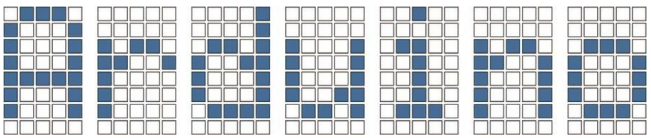
Principes intéressants

Avant d'utiliser l'afficheur LCD, voici quelques principes importants et intéressants à connaître. Comment un afficheur de ce type fonctionne-t-il ? Nous avons déjà vu que les différents caractères étaient composés à partir d'une matrice de points (Dot-Matrix). *Dot* signifie *point* et se trouve être le plus petit élément représentable dans cette matrice. Tout caractère est construit avec une matrice de points 5 × 8.



◀ Figure 15-2
La matrice de points 5 × 8 de l'afficheur LCD

Un emploi judicieux des différents points permet de générer les caractères les plus divers. La figure suivante montre le mot Arduino et les différents points à partir desquels les lettres sont composées.



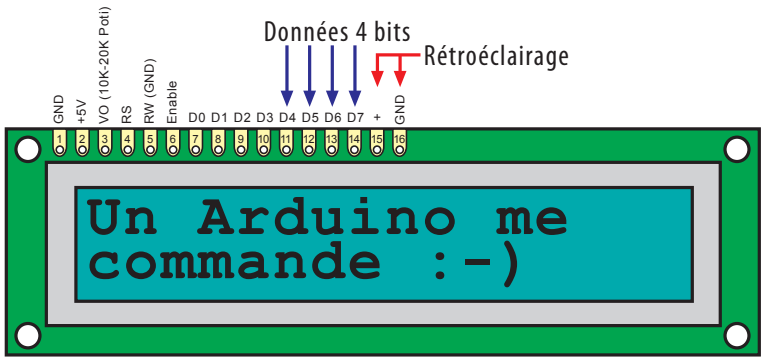
◀ Figure 15-3
Le mot Arduino composé à partir des différents points

La commande de l'afficheur est parallèle, autrement dit tous les bits de données sont envoyés en même temps au contrôleur. Il existe deux modes différents (4 bits et 8 bits), le mode 4 bits étant le plus utilisé parce qu'un nombre moindre de lignes de données doit être relié à l'afficheur, ce qui fait diminuer le coût.

Mais peut-être vous dites-vous que si nous utilisons 4 bits au lieu de 8, nous aurons un débit de données moindre et nous pourrons transmettre moins d'informations différentes. Comment faire alors ?

En réalité, cela fonctionne sans diminution du volume d'informations. En mode 4 bits, les 8 bits d'informations à transmettre sont simplement scindés en deux moitiés : l'une contenant les quatre premiers bits, et l'autre contenant les quatre derniers bits. Un nombre binaire de 4 bits est appelé *nibble* (quartet) dans le traitement des données. Les 4 bits d'un nibble sont transmis en parallèle, et les deux nibbles d'un octet en série. Surtout, ne vous en faites pas. Le mode 4 bits est certes plus lent que le mode 8 bits, mais ça n'a ici aucune importance. Venons-en maintenant au module d'affichage LCD Hitachi HDD44780, à son brochage et aux branchements nécessaires. Il existe deux variantes différentes : celle à 16 broches qui possède un rétroéclairage, et celle à 14 broches qui n'en a pas besoin.

Figure 15-4 ▶
Branchements du module d'affichage



Sur les huit lignes de données, seules les quatre lignes supérieures (D4 à D7) sont nécessaires. Le tableau 15-2 donne l'affectation des broches et leur signification.

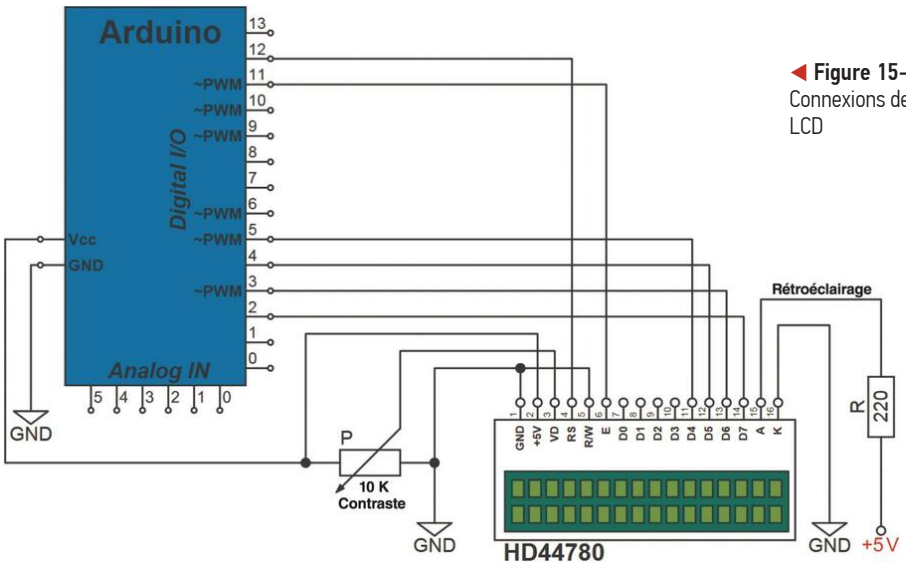
Broche LCD	Broche Arduino	
1	GND	Masse
2	+ 5V	+ 5 V
3	–	Réglage du contraste par un potentiomètre de 10K ou 20K
4	12	RS (Register Select)
5	GND	RW (Read/Write)/connecté à la masse (HIGH : Read/LOW : Write)
6	11	E (Enable)
11	5	Ligne de données D4
12	4	Ligne de données D5
13	3	Ligne de données D6
14	2	Ligne de données D7
15	–	Anode (+)/via résistance série de 220 Ω !
16	GND	Cathode (-)

◀ Tableau 15-2
Affectation des broches LCD pour la variante à 16 broches

L'objectif du premier sketch LCD est de faire apparaître à l'écran la phrase « Un Arduino me commande ».

Schéma

Le schéma de commande de l'afficheur LCD ne nécessite pas beaucoup de composants supplémentaires :



◀ Figure 15-5
Connexions de l'afficheur LCD

Réalisation du circuit

La réalisation du circuit est rapide sur une petite plaque de prototypage.

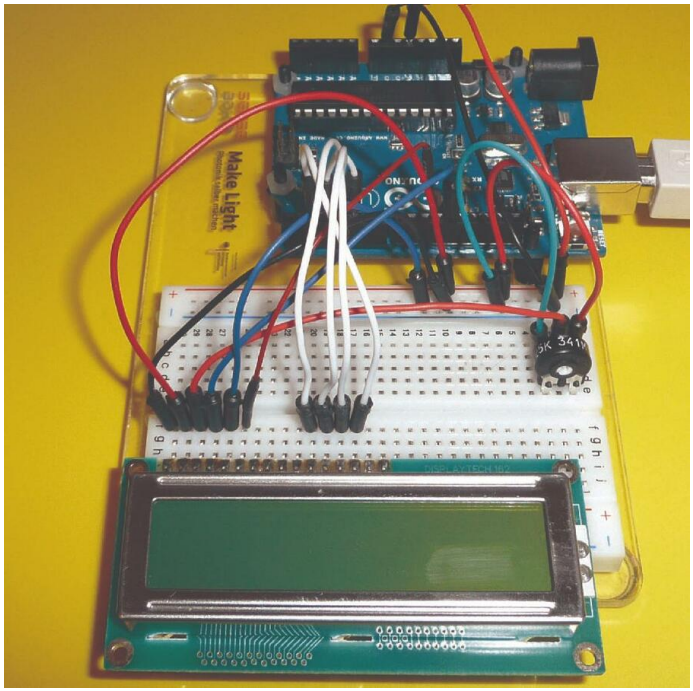


Figure 15-6 ▶
Réalisation du circuit de commande de l'afficheur LCD

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Revue de code

Le sketch suivant permet d'afficher un texte de 2 lignes sur l'écran LCD.

```
#include <LiquidCrystal.h>
#define RS 12   // Register Select
#define E 11    // Enable
#define D4 5    // Ligne de données 4
#define D5 4    // Ligne de données 5
#define D6 3    // Ligne de données 6
#define D7 2    // Ligne de données 7
#define COLS 16 // Nombre de colonnes
#define ROWS 2  // Nombre de lignes
LiquidCrystal lcd(RS, E, D4, D5, D6, D7); // Instanciation de l'objet
```

```
void setup() {
  lcd.begin(COLS, ROWS);        // Nombres de lignes et de colonnes
  lcd.print("Un Arduino me");   // Affichage du texte
  lcd.setCursor(0, 1);          // Passer à la 2e ligne
  lcd.print("commande :-)");    // Affichage du texte
}
void loop() { /* vide */ }
```

Examinons la signification de ce sketch.

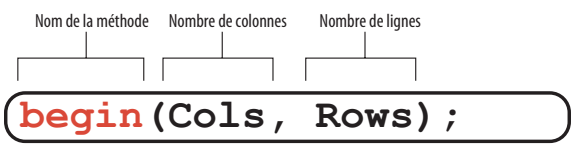
La bibliothèque `LiquidCrystal` doit être incorporée afin de pouvoir utiliser la fonctionnalité des commandes de l'afficheur LCD. Les paramètres suivants doivent être communiqués au constructeur pour générer un objet LCD :

- broche Register Select (`RS`) ;
- broche Enable (`E`) ;
- broches des lignes de données `D4` à `D7`.

```
LiquidCrystal lcd(RS, E, D4, D5, D6, D7); // Instanciation de l'objet
```

La classe `LiquidCrystal` met une série de méthodes à disposition, car on ne peut envoyer un texte à l'afficheur LCD avec le seul constructeur. Pour que ce soit possible, il nous faut transmettre à l'objet afficheur quelques informations supplémentaires pour poursuivre l'initialisation. Les afficheurs LCD diffèrent quant au nombre de colonnes ou de lignes, et ce sont précisément ces informations qu'il lui faut. On voit bien que le constructeur ne dispose pas de tout pour une initialisation complète. Une méthode est ici nécessaire.

Méthode LCD : begin



◀ **Figure 15-7**
Méthode LCD : begin

La méthode `begin` communique les nombres de colonnes et de lignes à l'objet LCD. Tout est alors prêt pour envoyer un texte.

Méthode LCD : print

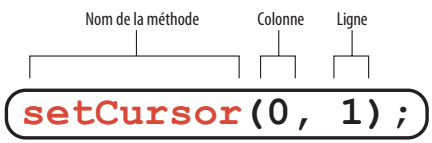
◀ **Figure 15-8**
Méthode LCD `print`

La méthode `print` indique à l'objet LCD ce qui doit être affiché à l'écran. Elle est comparable à celle du moniteur série.

Quand aucune indication n'est donnée sur la position du texte à afficher, celui-ci se place au début de la première ligne. Comme vous pouvez le voir dans l'exemple, une autre ligne est occupée par du texte. Venons-en maintenant à la troisième méthode importante.

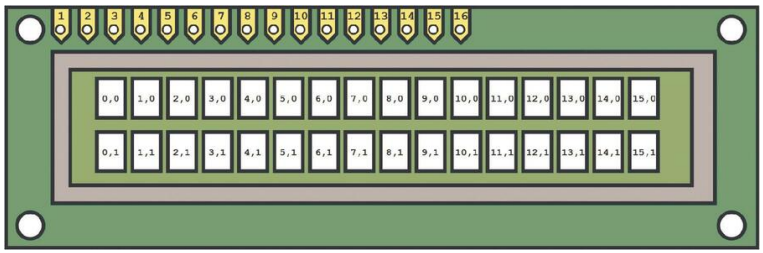
Méthode LCD : setCursor

Figure 15-9 ▶
Méthode LCD `setCursor`



La méthode `setCursor` permet de positionner le curseur à l'endroit où le texte suivant doit commencer. lle est ici aussi – et comment pourrait-il en être autrement – basée sur zéro, autrement dit la première ligne ou colonne est pourvue de l'index 0. Pour atteindre la deuxième ligne, vous devez – comme c'est le cas ici – utiliser la valeur 1. La figure suivante peut vous aider à positionner l'affichage.

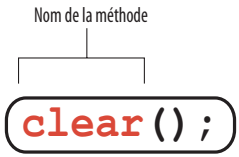
Figure 15-10 ▶
Coordonnées des différentes lignes accessibles avec `setCursor`



Avant d'oublier : vous pouvez naturellement tout effacer jusqu'au dernier caractère avec la méthode `clear`. La méthode suivante est utilisée pour ce faire :

Méthode LCD : clear

Figure 15-11 ▶
Méthode LCD `clear`



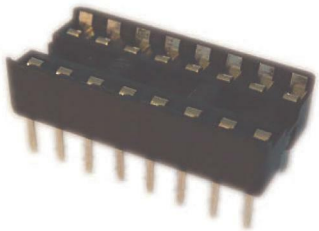
Elle n'a pas de paramètre, efface tous les caractères de l'afficheur et positionne le curseur sur la coordonnée `0,0` dans le coin supérieur gauche.

VARIANTES DU HD44780

Dans certaines variantes du HD44780, on peut brancher le rétroéclairage sur +5 V sans résistance série ; dans d'autres, une résistance dimensionnée en conséquence est nécessaire. Regardez la fiche technique avant de brancher la tension d'alimentation. Vous pouvez au pire laisser tomber le rétroéclairage. Si c'est trop sombre, vous pourrez toujours augmenter le contraste de manière à pouvoir lire quand même l'affichage.

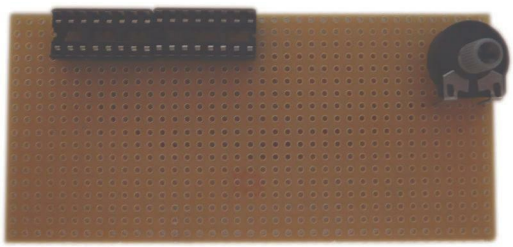
Jeu : deviner un nombre

Quoi de mieux que le jeu où il faut deviner des nombres pour une réalisation avec l'afficheur LCD ? Si ça fonctionne, vous n'aurez plus besoin d'un ordinateur et gagnerez en indépendance grâce à l'unité d'affichage LCD. J'ai fixé, pour le besoin de la réalisation, le LCD sur une carte de circuit imprimé perforée, sur laquelle deux supports de circuit intégré à 16 broches sont posés l'un à côté de l'autre.



◀ Figure 15-12
Support de circuit intégré à 16 broches

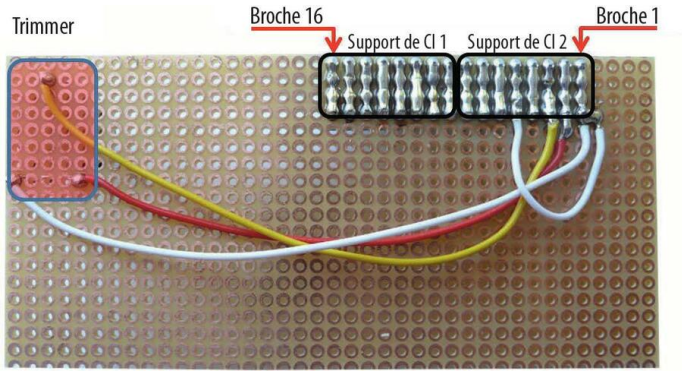
Un support comparable – mais avec plus de broches bien sûr – se trouve sur votre carte Arduino et maintient le microcontrôleur en position. De tels connecteurs sont vraiment utiles, car si un circuit intégré vient à griller pour de bon, plus besoin de le dessouder péniblement. Il suffit de le remplacer. La carte mesure 10 × 5 cm.



◀ Figure 15-13
Carte-support pour l'afficheur LCD

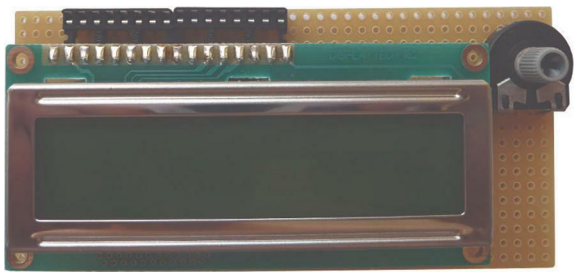
Comme vous pouvez le constater, j'ai également placé dessus le potentiomètre trimmer pour le réglage du contraste. De l'autre côté de la carte, on voit comment j'ai relié entre elles les différentes broches des supports de circuit intégré. Les broches opposées ont toutes été reliées par plusieurs points de soudure.

Figure 15-14 ▶
Face soudée de la carte-support pour l'afficheur LCD



Un *câblage volant* est parfois suffisant. La figure suivante montre l'afficheur LCD déjà fixé sur la carte. Ses broches sont insérées dans les contacts de la rangée inférieure des deux supports de circuit intégré. La rangée supérieure servira plus tard pour la connexion au shield.

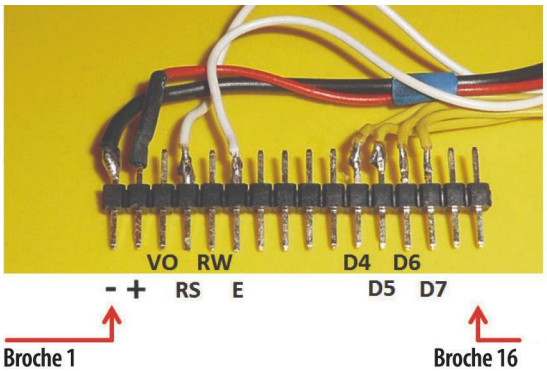
Figure 15-15 ▶
Afficheur LCD sur sa carte-support



Il ne manque plus maintenant que les lignes de raccordement à votre clavier analogique. Les liaisons passent par les barrettes à broches que vous connaissez bien. Il vous faut :

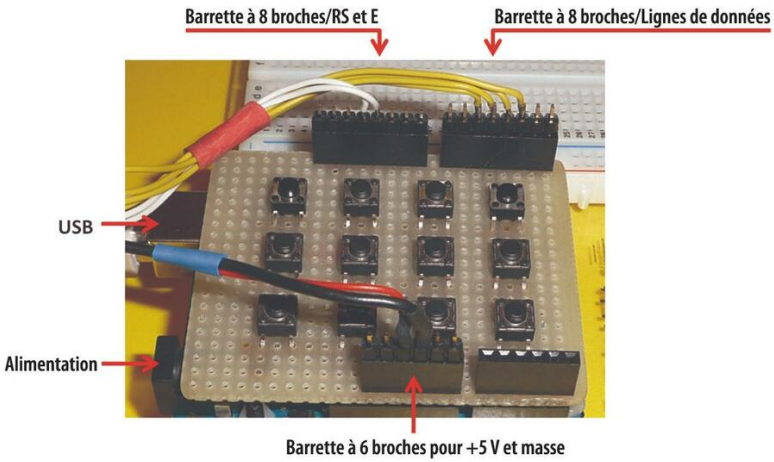
- 1 barrette à 16 broches ;
- 2 barrettes à 8 broches ;
- 1 barrette à 6 broches.

La barrette à 16 broches est raccordée à la carte-support sur laquelle se trouve l'afficheur LCD. La figure suivante illustre les lignes de raccordement analogique.



◀ Figure 15-16
Barrette à 16 broches

Les deux barrettes à 8 broches et la barrette à 6 broches sont branchées sur le clavier.



◀ Figure 15-17
Clavier analogique avec ses trois barrettes à broches

En utilisant les couleurs et l'affectation des broches indiquée, vous ne devriez pas avoir de problème pour construire le petit faisceau de câbles avec les barrettes à broches. La figure 15-18 montre à nouveau les trois composants, à savoir la carte-support avec l'afficheur LCD, la carte Arduino et le shield du clavier superposé, tous reliés entre eux.

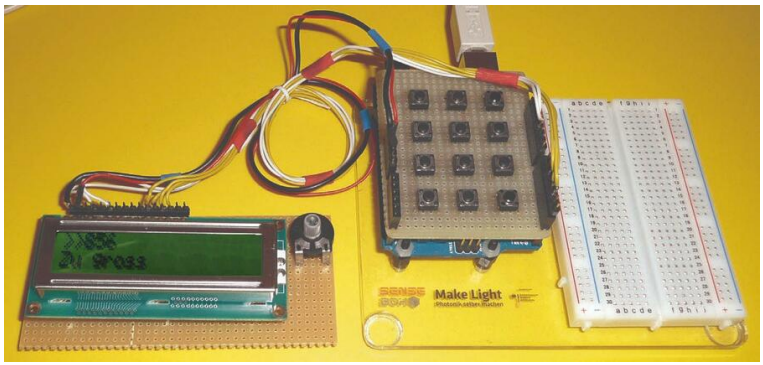


Figure 15-18 ▶
Réalisation du circuit complet pour le jeu des nombres à deviner

Vous pouvez naturellement inscrire les différentes touches au marqueur sur la carte. Mais le mieux est de fabriquer vous-même un clavier à film à l'aide d'une plastifieuse.

ATTENTION AUX COURTS-CIRCUITS !

Si vous soudez sur les barrettes à broches des fils d'alimentation ou de masse qui se trouvent directement l'un à côté de l'autre, vous risquez fort d'avoir un jour ou l'autre, par déplacement, un court-circuit entre ces contacts ou les fils avoisinants. Aussi ai-je pris soin de mettre chaque soudure sous gaine thermorétractable pour l'isoler. Pour plus de sécurité, vous pouvez le faire pour toutes les broches auxquelles des fils sont connectés :

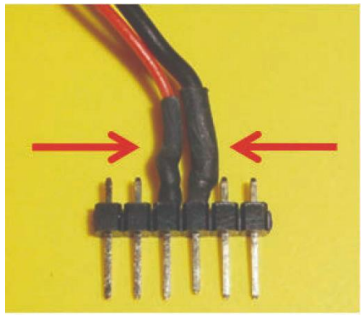


Figure 15-19 ▶
Barrette à 6 pôles avec deux morceaux de gaine thermorétractable (flèches rouges)

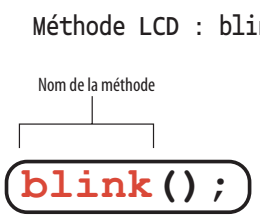
Voici maintenant le code complet, qui est déjà un peu plus conséquent.

```
#include <LiquidCrystal.h>
#include "MyAnalogKeyPad.h"

#define analogPinKeyPad A0 // Définition de la broche analogique
#define MIN 10             // Limite inférieure du nombre aléatoire
#define MAX 1000           // Limite supérieure du nombre aléatoire
#define RS 12              // Broche Register Select du LCD
#define E 11               // Broche Enable du LCD
#define D4 5               // Ligne de données LCD broche 4
#define D5 4               // Ligne de données LCD broche 5
#define D6 3               // Ligne de données LCD broche 6
#define D7 2               // Ligne de données LCD broche 7
#define COLS 16            // Nombre de colonnes LCD
#define ROWS 2             // Nombre de lignes LCD

int arduinoNumber, tries;  // Le nombre généré, nombre d'essais
char yourNumber[5];        // Nombre à 5 chiffres maxi
byte place;
MyAnalogKeyPad myOwnKeyPad(analogPinKeyPad); // Instanciation clavier
LiquidCrystal lcd(RS, E, D4, D5, D6, D7);    // Instanciation LCD

void setup() {
  myOwnKeyPad.setDebounceTime(500); // Régler le temps du rebond à 500 ms
  lcd.begin(COLS, ROWS);            // Nombres de lignes et de colonnes
  lcd.blink();                      // Faire clignoter le curseur
 startSequence();                   // Appel de la séquence de démarrage
}

void loop() {
  char myKey = myOwnKeyPad.readKey(); // Lecture de la touche pressée
  if(myKey != KEY_NOT_PRESSED) { // Interrogation si une touche quelconque
                                 // est pressée
    yourNumber[place] = myKey;
    place++;
    lcd.print(myKey);          // Afficher la touche sur le LCD
  }
  if(place == int(log10(MAX))+1) {
    tries++;
    int a = atoi(yourNumber);
    if(a == arduinoNumber) {
      lcd.clear();             // Effacer écran LCD
      lcd.print("Exact !!!");  // Affichage sur LCD
      lcd.setCursor(0, 1);     // Positionnement curseur sur 2e ligne
      lcd.print("Essai : " + String(tries));
      delay(4000);             // Attendre 4 secondes
      tries = 0;               // Remise à zéro du nombre d'essais
      startSequence();         // Appel de StartSequence
    }
    else if(a < arduinoNumber) {
      lcd.setCursor(0, 1);     // Positionnement curseur sur 2e ligne
      lcd.print("Trop petit"); // Affichage sur LCD
```

```
      lcd.setCursor(0, 0);      // Positionnement curseur sur 1re ligne
    }
    else {
      lcd.setCursor(0, 1);      // Positionnement curseur sur 2e ligne
      lcd.print("Trop grand"); // Affichage sur LCD
      lcd.setCursor(0, 0);      // Positionnement curseur sur 1re ligne
    }
    lcd.setCursor(2, 0);  // Positionnement curseur sur 3e emplacement
                          // de la 1re ligne
    place = 0;
  }
}

int randomNumber(int minimum, int maximum) {
  randomSeed(analogRead(5));
  return random(minimum, maximum + 1);
}

void startSequence() {
  arduinoZahl = randomNumber(MIN, MAX); // Générer le nombre à deviner
  lcd.clear();                 // Effacer écran LCD
  lcd.print("Devine un nombre"); // Affiche sur LCD
  lcd.setCursor(0, 1);          // Positionnement curseur sur 2e ligne
  lcd.print("de " + String(MIN) + " - " + String(MAX));
  delay(4000);              // Attendre 4 secondes
  lcd.clear();              // Effacer écran LCD
  lcd.print(">>");          // Affiche sur LCD
}
```

Je ne souhaite pas trop m'étendre sur le sketch pour l'instant. J'ai fait en sorte que l'affichage ait lieu sur le LCD. J'ai ajouté aussi une méthode qui affiche un curseur clignotant à l'écran (voir figure 15-20).

Figure 15-20 ▶
Méthode LCD `blink`



`blink` est appelée une seule fois dans la fonction `setup` et fait clignoter un curseur à l'endroit où on va écrire. Quand on regarde le début du sketch, on voit qu'il est tout à fait possible d'incorporer plusieurs bibliothèques dans un projet. Il n'y a théoriquement aucune limite. La mémoire flash finit quand même à un moment par laisser entendre qu'elle est pleine et qu'aucun code ne peut plus être ajouté.

Étudions de près la ligne

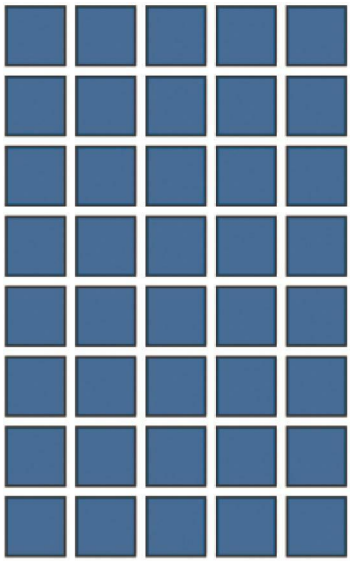
```
lcd.print("de" + String(MIN) + " - "+ String(MAX));
```

Dans cette ligne, on affiche des chaînes de caractères et on utilise l'opérateur `+`. Mais comment fait-on pour additionner des chaînes de caractères ? Ça ne marche qu'avec des nombres, non ?

Évidemment ! Seules des valeurs mathématiques peuvent être additionnées. L'opérateur `+` ne peut évidemment rien additionner dans le cas de chaînes de caractères. Comment le pourrait-il ? Les différentes chaînes de caractères sont simplement réunies en une seule. On dit aussi qu'elles sont *concaténées*. Si maintenant, comme dans notre sketch, des valeurs numériques font partie de la chaîne de caractères à afficher, elles doivent être préalablement converties en une `string`. C'est la fonction `string` qui s'en charge, comme dans `String(MIN)`.

Définir des caractères personnels

Les caractères standards pour l'affichage d'informations sont bien connus. Toutefois, il est aussi possible de définir ses propres signes et symboles. Souvenez-vous de la matrice composée de 5 × 8 pixels.



◀ Figure 15-21
Matrice de pixels

Supposons que vous vouliez afficher un smiley gai et un smiley triste. Il faudrait alors changer la matrice pour chaque icône. La figure 15-22 montre la solution que je vous propose.

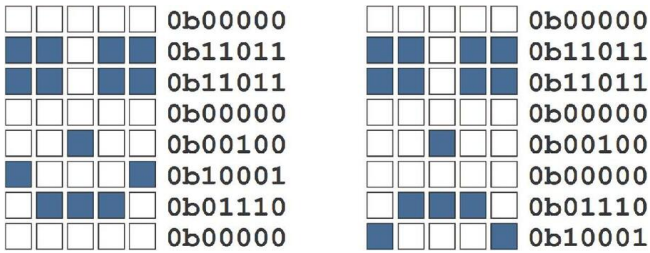


Figure 15-22 ▶
Matrice de pixels pour les deux smileys

À droite des icônes, vous pouvez voir les valeurs binaires : un 1 correspond à un pixel visible. Le mode d'écriture avec le préfixe `0b` indique qu'il s'agit d'un nombre binaire. Examinons le sketch commandant l'affichage des deux smileys. Par manque de place, je vous épargne les lignes d'introduction consacrées à la définition des valeurs ou à l'instanciation de l'objet :

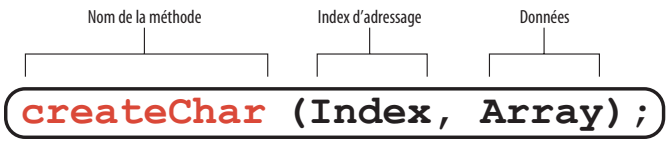
```
// Smiley joyeux
byte customChar0[8] = {
  0b00000,
  0b11011,
  0b11011,
  0b00000,
  0b00100,
  0b10001,
  0b01110,
  0b00000
};

// Smiley triste
byte customChar1[8] = {
  0b00000,
  0b11011,
  0b11011,
  0b00000,
  0b00100,
  0b00000,
  0b01110,
  0b10001
};

void setup() {
  lcd.createChar(0, customChar0);
  lcd.createChar(1, customChar1);
  lcd.begin(COLS, ROWS);
  lcd.write((byte)0);
  lcd.write((byte)1);
```

```
}

void loop() { /* vide */ }
```

Pour plus d'informations sur la génération d'une matrice avec le code correspondant, consultez le site Internet suivant :

https://omerk.github.io/lcdchargen/

Les deux tableaux contiennent les informations relatives aux pixels des deux smileys. La méthode `createChar` est utilisée pour la création des nouvelles icônes :



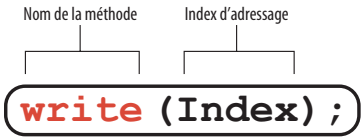
La ligne

```
lcd.createChar(0, customChar0);
```

donne accès au tableau `customChar0` avec l'index 0 dont nous aurons besoin plus tard pour la méthode `write`. Pour plus d'informations, consultez la page web :

https://www.arduino.cc/en/Reference/LiquidCrystalCreateChar

La méthode `write` affiche un caractère :



La ligne

```
lcd.write((byte)0);
```

nécessite quelques explications. Elle ne définit pas simplement la valeur d'index du tableau de caractères, mais elle est précédée d'un type de données entre parenthèses. Ce procédé appelé *casting* oblige presque la donnée à prendre le type de données précisé. Elle peut donc être considérée comme appartenant au type de données `byte`, comme le veut la bibliothèque. Cela ne me paraît pas très heureux, mais c'est ainsi et l'on n'y peut rien. Pour plus d'informations, consultez le lien suivant :

https://www.arduino.cc/en/Reference/LiquidCrystalWrite

Un afficheur LCD multiligne

Si un afficheur à deux lignes vous paraît insuffisant et si vous préférez une commande plus conviviale, alors le module suivant vaut certainement que vous vous y attardiez.

Figure 15-23 ▶
Afficheur LCD à 4 lignes (commande par bus I²C)

Si vous examinez cette photo attentivement, vous constatez que quatre fils seulement sont raccordés à l'écran. Avant, il y en avait bien plus. Comment cela se fait-il ? Manquerait-il quelque chose ?

Bien vu ! L'afficheur LCD que j'utilise ici est commandé via le bus I²C. Nous verrons bientôt de quoi il s'agit précisément. Pour l'instant, sachez que deux fils de commande suffisent. Les deux autres fils servent à l'alimentation électrique et il est donc difficile de s'en passer. Si vous souhaitez trouver plus d'informations sur ce module sur Internet, saisissez les critères de recherche suivants :

```
ywrobot arduino lcm1602
```

Le code du sketch permettant d'afficher le message illustré est relativement simple :

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);

void setup() {
  lcd.begin(20,4);
  lcd.backlight();
  lcd.setCursor(0,0); lcd.print("Bonjour !");
  lcd.setCursor(0,1); lcd.print("Cet écran offre");
  lcd.setCursor(0,2); lcd.print("beaucoup plus");
  lcd.setCursor(0,3); lcd.print("de possibilités !");
}

void loop() { /* vide */ }
```

Les trois premières lignes sont intéressantes :

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
```

Elles incorporent la fonctionnalité d'utilisation du bus I^2C et de la bibliothèque I^2C pour l'écran LCD. La dernière ligne contient divers paramètres, comme les adresses de commande de l'afficheur. Vous en saurez plus bientôt.

Exercice complémentaire

Réfléchissez un peu à une serrure à code de sécurité, du type de celles installées aux entrées des zones sensibles. Un code à plusieurs chiffres doit être saisi pour ouvrir la porte. On est bien entendu informé en cas de saisie erronée que le code chiffré composé est trop bas ou trop élevé. Vous pouvez par exemple brancher un servomoteur désengageant un pêne du système de fermeture en cas de code correct. Il faut attendre un certain temps, par exemple trois minutes, après avoir tapé trois fois de suite un code erroné. Créez un contrôle d'accès à votre chambre pour décourager les colocataires ou proches trop curieux.

Problèmes courants

Si, après avoir raccordé le LCD et chargé le sketch, vous ne voyez rien à l'écran, vérifiez les points suivants.

- Le câblage est-il correct ?
- Pas de court-circuit ?
- Le trimmer du contraste est-il correctement branché ? Augmentez le cas échéant le contraste jusqu'à ce que vous puissiez voir quelque chose à l'écran.

Qu'avez-vous appris ?

- Vous avez raccordé pour la première fois un élément d'affichage, capable non seulement de clignoter comme une LED, mais aussi d'afficher des nombres et du texte.
- La bibliothèque `LiquidCrystal` vous a permis de commander facilement un LCD avec un contrôleur HDD44780.
- Vous avez ensuite clairement transposé le jeu des nombres à deviner.
- D'autres types de LCD vous ont été présentés pour vos expériences à venir, de telle sorte que vos créations puissent être sans limite.
- Je vous ai présenté un afficheur LCD dont la commande ne nécessite que deux fils, ce qui est rendu possible par le bus I^2C.

16 Le moteur pas-à-pas

Si vous voulez obtenir un positionnement précis et répétés à l'aide d'un moteur, alors les moteurs à courant continu ne sont pas adaptés. L'axe tourne quand une tension est appliquée et cesse de tourner quand la tension n'est plus appliquée. Une certaine inertie est presque inévitable. D'autres moteurs sont mieux adaptés.

Encore plus de mouvement

Un servomoteur permet de transformer le courant électrique en mouvement. Cependant, son rayon d'action demeure limité, même si des modifications peuvent être entreprises pour combler ce manque. Mais il s'avère suffisant pour la plupart des applications. Le moteur pas-à-pas s'impose en revanche si une plus grande liberté d'action est nécessaire. Par souci d'économie, vous pouvez essayer d'en récupérer un sur un vieil appareil quelconque :

- imprimante ;
- scanner à plat ;
- lecteur de CD/DVD ;
- imprimante 3D ;
- lecteur de disquette de 3,5 pouces.

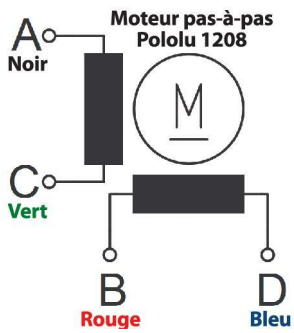
Ces lecteurs possèdent un petit moteur pas-à-pas, de type PL15S-020 la plupart du temps, qui entraîne un petit chariot sur lequel se trouve la tête d'écriture/lecture. La figure suivante montre une unité de ce genre relativement bon marché.

Figure 16-1 ▶
Moteur pas-à-pas 1208 de Pololu

Le moteur pas-à-pas dispose de 4 bornes, sur lesquelles nous allons revenir en détail. Le schéma de raccordement d'un moteur pas-à-pas est illustré sur la figure suivante. Attention aux repères couleur des fils associés par paire.

- noir + vert
- rouge + bleu

Figure 16-2 ▶
Branchements du moteur pas-à-pas



Vu que ce moteur dispose de 4 bornes, il s'agit d'un moteur pas-à-pas bipolaire. Sa tension de fonctionnement est de 10 V avec 500 mA/bobine. Vous trouverez de plus amples informations à l'adresse suivante :

https://www.pololu.com/product/1208

Pour mettre le moteur en marche, les bornes en question doivent recevoir certaines impulsions dans un ordre chronologique bien déterminé.

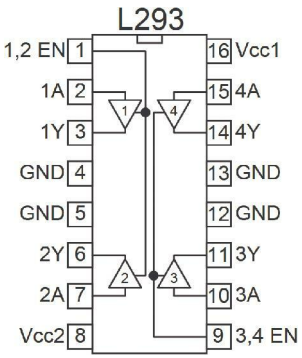
Figure 16-3 ▶
Séquence de commande pour moteur pas-à-pas 1208

dans le sens des aiguilles d'une montre ↓

	Bornes			
PAS	A	C	B	D
0	**HIGH**	LOW	**HIGH**	LOW
1	LOW	**HIGH**	**HIGH**	LOW
2	LOW	**HIGH**	LOW	**HIGH**
3	**HIGH**	LOW	LOW	**HIGH**

Quand on écrit un sketch pour traiter successivement les pas 1 à 4 et envoyer les niveaux LOW ou HIGH correspondant au moteur pas-à-pas, ce dernier tourne dans le sens horaire. Quand l'ordre des pas est inversé,

le sens est antihoraire. Il y a une chose importante que je n'ai pas encore dite : on ne peut pas se contenter de raccorder le moteur pas-à-pas aux sorties numériques, car elles seraient alors tant sollicitées que la carte en pâtirait. Aussi utilise-t-on ici un circuit de commande de moteur de type L293 (voir figure 16-4).



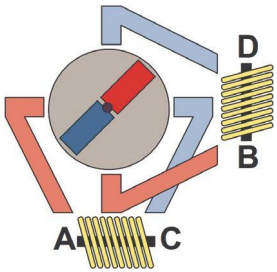
◀ Figure 16-4
Commande de moteur de type L293

Les petits triangles sont le symbole du circuit driver nécessaire pour fournir la puissance dont un moteur raccordé a besoin pour fonctionner. Les bornes A du circuit intégré sont les entrées et celles Y sont les sorties. Chaque paire de drivers a une borne de validation commune, libellée *1,2EN* ou *3,4EN* (*EN* pour *enable*, ou permettre). Ce circuit de commande de moteur peut fournir un courant de 600 mA par sortie. Les circuits de commande suivants sont capables de délivrer un courant plus élevé :

- SN754410 (1 ampère) ;
- L298 (2 ampères).

Expliquons à présent le fonctionnement d'un moteur pas-à-pas. Il existe différents types de moteurs pas-à-pas : unipolaires et bipolaires, ces derniers étant plus faciles à commander. C'est aussi pour cette raison que je m'en sers dans ce montage. Un moteur bipolaire pas-à-pas possède deux bobines dont les quatre raccordements sortent du boîtier. Un moteur unipolaire pas-à-pas est construit sur le même modèle, à la différence que chaque bobine possède en son milieu un raccordement qui sort du boîtier. Un changement de direction du mouvement est obtenu en inversant les pôles du champ magnétique c'est-à-dire du sens du courant. Sur la figure suivante, vous pouvez voir les deux bobines avec les fils de raccordement A/C et B/D, qui bougent en fonction de la polarité du rotor au milieu :

Figure 16-5 ▶
Schéma du moteur pas-à-pas bipolaire



Le symbole du moteur pas-à-pas bipolaire est le suivant :

Figure 16-6 ▶
Symbole du moteur pas-à-pas bipolaire

Vous trouverez de plus amples informations à l'adresse suivante :

https://fr.wikipedia.org/wiki/Moteur_pas_à_pas

Composants nécessaires

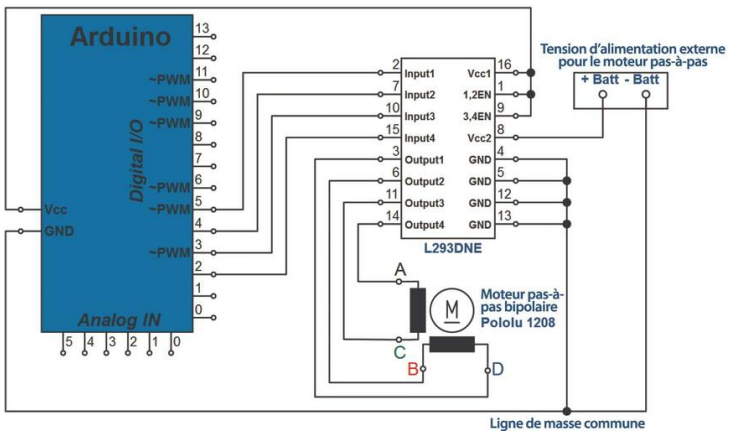
Ce montage nécessite les composants suivants.

Tableau 16-1 ▶
Liste des composants

Composant	
1 moteur pas-à-pas bipolaire, par ex. Pololu 1208	
1 circuit de commande de moteur L293DNE	
1 clip pour pile de 9 V	

Schéma

Sur le schéma, vous pouvez voir le circuit de commande L293DNE du moteur pas-à-pas raccordé à une source de tension externe fournie par une pile de 9 V.



◀ Figure 16-7
Schéma de commande du moteur pas-à-pas

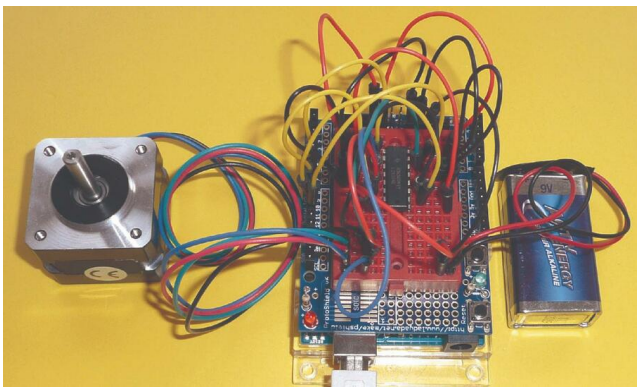
Notez que la masse de la source de tension externe doit être raccordée à la masse de la carte Arduino.

ATTENTION AU RACCORDEMENT DU PÔLE + !

Le pôle + de la carte Arduino ne doit jamais être raccordé à la source de tension externe. Cela détruirait votre carte Arduino ou le port USB de votre ordinateur !

Réalisation du circuit

J'ai réalisé le circuit avec un Proto Shield, car tout tient à merveille sur cette mini-plaque d'essais :



◀ Figure 16-8
Réalisation du circuit de commande du moteur pas-à-pas

Pas de panique si vous ne trouvez pas de *Pololu 1208* ! Vous pouvez en fait prendre pratiquement n'importe quel moteur pas-à-pas bipolaire. Il vous suffit de trouver la fiche technique correspondante sur Internet pour en connaître les spécifications. Attention cependant au courant consommé par le moteur pas-à-pas en service ; comparez-le à celui pour le circuit de commande de moteur utilisé ici. Il ne doit en aucun cas dépasser 600 mA par borne. Faute de quoi, vous devez prendre soit un autre moteur pas-à-pas, soit un autre circuit de commande.

Si vous regardez le schéma, vous pouvez remarquer que la source de tension externe, d'après les données du moteur pas-à-pas, doit être de 5 V. Mais vous pouvez utiliser une pile de 9 V, comme moi, car cela fonctionne parfaitement. En cas d'utilisation prolongée, utilisez de préférence un bloc secteur. Nous en arrivons à la programmation du sketch. Que dois-je ajouter ? Il existe une bibliothèque pour cet élément. Alors, pourquoi réinventer la roue ? La bibliothèque est très bien fournie, car elle permet de commander différents modèles de moteurs pas-à-pas. Plusieurs possibilités s'offrent à nous :

- variation de la cadence ;
- saisie du déplacement ;
- les valeurs positives correspondent à une rotation vers la droite, tandis que les valeurs négatives produisent une rotation vers la gauche.

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Revue de code

Le sketch exploite les possibilités mentionnées, c'est-à-dire le réglage de la cadence, du déplacement et du sens de rotation :

```
#include <Stepper.h>
#define NOIR   2
#define VERT   3
#define ROUGE  4
#define BLEU   5
#define SPEED  100 // 100 tr/min
const int stepsPerRevolution = 200; // Pas par tour

Stepper myStepper(stepsPerRevolution, BLEU, ROUGE, VERT, NOIR);

void setup() {
  myStepper.setSpeed(SPEED); // Réglage de la vitesse
}
```

```
void loop() {
  myStepper.step(stepsPerRevolution);
  delay(500); // Courte pause
  myStepper.step(-stepsPerRevolution);
  delay(500); // Courte pause
}
```

Examinons la signification de ce sketch.

La bibliothèque Stepper est liée au sketch à l'aide de l'instruction `include` :

```
#include <Stepper.h>
```

Cela vous permet d'appeler diverses méthodes ou d'effectuer des configurations fondamentales dans le constructeur (nombre de pas par tour et brochages) qui ne sont plus modifiées dans le sketch. L'objet Stepper est créé par l'instanciation de la classe dans la ligne :

```
Stepper myStepper(stepsPerRevolution, BLEU, ROUGE, VERT, NOIR);
```

Le constructeur possède ici cinq paramètres qui doivent être communiqués dans l'ordre indiqué. Nous en arrivons à la première méthode qui définit la vitesse du moteur pas-à-pas. Nous utilisons `setSpeed` :

```
myStepper.setSpeed(SPEED);
```

La ligne

```
myStepper.step(stepsPerRevolution);
```

met le moteur en marche par l'activation de la méthode `step`. L'inversion du signe de l'argument transmis permet d'inverser le sens de rotation. Une courte pause doit être marquée lors du changement de sens. La bibliothèque stepper est programmée de façon à pouvoir être utilisée avec des moteurs pas-à-pas ayant différents nombres de broches. Nous avons déjà vu ensemble les bases de la programmation orientée objet et la signification d'une surcharge. Voici un extrait du fichier Stepper.h :

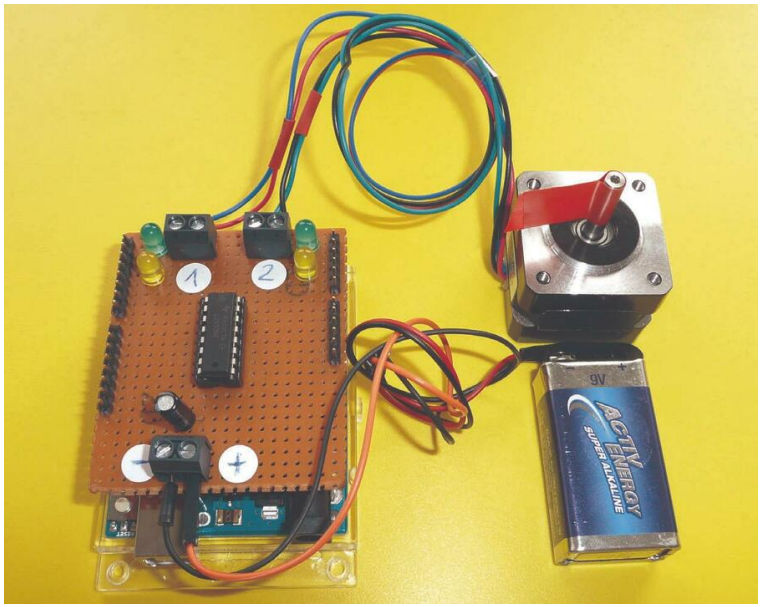
```
class Stepper {
  public:
    // constructors:
    Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2);
    Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2,
                                 ➥ int motor_pin_3, int motor_pin_4);
    Stepper(int number_of_steps, int motor_pin_1, int motor_pin_2,
                                 ➥ int motor_pin_3, int motor_pin_4,
                                 ➥ int motor_pin_5);
    // ...
};
```

Vous pouvez voir ici trois constructeurs qui portent tous le même nom, mais qui se différencient par leur signature, c'est-à-dire leur nombre de paramètres. Ainsi, lors de l'instanciation, le compilateur saura précisément combien votre moteur pas-à-pas possède de broches (2, 4 ou 5).

Construire un shield pour moteur

Vous pouvez évidemment acheter un shield pour moteur prêt à l'emploi pour commander des moteurs à courant continu ou pas-à-pas, mais j'ai décidé d'en construire un moi-même, comme celui illustré sur la figure suivante.

Figure 16-9 ▶
Shield pour moteur



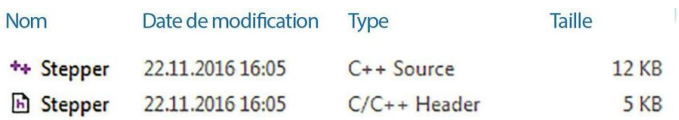
Quelques composants suffisent pour réaliser ce type de montage.

Commande de servomoteurs

Si vous souhaitez commander des servomoteurs avec la bibliothèque Stepper, jetez un œil à la bibliothèque suivante que vous trouverez à l'emplacement indiqué ci-dessous :

```
C:\Program Files (x86)\Arduino\libraries\Stepper\src
```

Vous y trouverez le fichier d'en-tête, ainsi que le fichier cpp :



Nom	Date de modification	Type	Taille
Stepper	22.11.2016 16:05	C++ Source	12 KB
Stepper	22.11.2016 16:05	C/C++ Header	5 KB

Programmation d'un sketch personnalisé

Si vous souhaitez savoir comment programmer vous-même ce type de commande, alors examinez le code suivant. J'expliquerai le sketch directement dans la foulée. Les broches ont été définies à l'aide de l'instruction `define` et la séquence de commande correspond à celle présentée dans le tableau de la figure 16-3.

```
#define Stepper_A1 5 // Broche pour connexion A1
#define Stepper_A3 4 // Broche pour connexion A3
#define Stepper_B1 3 // Broche pour connexion B1
#define Stepper_B3 2 // Broche pour connexion B3
byte stepValues[5][4] = {{LOW,  LOW, LOW,  LOW},  // Moteur à l'arrêt
                         {HIGH, LOW, HIGH, LOW},  // Pas 1
                         {LOW, HIGH, HIGH, LOW},  // Pas 2
                         {LOW, HIGH, LOW, HIGH},  // Pas 3
                         {HIGH, LOW, LOW, HIGH}}; // Pas 4
```

La fonction `setup` programme les broches numériques comme sorties et active la fonction `action` qui possède quant à elle deux paramètres qui sont chargés de définir le nombre de pas, puis d'aménager une pause. La boucle `for` permet de faire tourner le moteur pas-à-pas vers la gauche et vers la droite, puis de le mettre hors courant.

```
void setup() {
  pinMode(Stepper_A1, OUTPUT);
  pinMode(Stepper_A3, OUTPUT);
  pinMode(Stepper_B1, OUTPUT);
  pinMode(Stepper_B3, OUTPUT);
  for(int i = 0; i < 10; i++){
    action(30, 2);   // 30 pas vers la droite avec pause de 2 ms
    action(-30, 10); // 30 pas vers la gauche avec pause de 10 ms
  }
  action(0, 0); // Mise hors courant
}
```

La fonction `loop` reste ici sans effet puisque la commande s'effectue exclusivement dans la fonction `setup`.

```
void loop(){ /* vide*/ }
```

Venons-en maintenant à la fonction `action`. Elle active les différentes étapes de la séquence de commande qui activent à leur tour la fonction `moveStepper` dans laquelle les valeurs du tableau sont précisées. Le sens de rotation du moteur pas-à-pas est défini par le signe de la variable `count`. La séquence de commande est activée une fois vers l'avant et une fois vers l'arrière :

```
void action(int count, byte delayValue){
  if(count > 0) // Rotation vers la droite
    for(int i = 0; i < count; i++)
      for(int sequenceStep = 1; sequenceStep <= 4; sequenceStep++)
        moveStepper(sequenceStep, delayValue);
  if(count < 0) // Rotation vers la gauche
    for(int i = 0; i < abs(count); i++)
      for(int sequenceStep = 4; sequenceStep > 0; sequenceStep--)
        moveStepper(sequenceStep, delayValue);
  if(count == 0) // Mise hors courant
    moveStepper(0, delayValue);
}

void moveStepper(byte s, byte delayValue){
  digitalWrite(Stepper_A1, stepValues[s][0]);
  digitalWrite(Stepper_A3, stepValues[s][1]);
  digitalWrite(Stepper_B1, stepValues[s][2]);
  digitalWrite(Stepper_B3, stepValues[s][3]);
  delay(delayValue); // Pause
}
```

Ce n'est évidemment que l'une des possibilités et il existe une multitude de variantes.

Problèmes courants

Si le moteur pas-à-pas ne bouge pas ou ne fait que bourdonner ou vibrer, vérifiez :

- que le câblage est correct ;
- qu'il n'y a pas de court-circuit ;
- que le moteur pas-à-pas ne change pas de position ou qu'il ne bourdonne pas ou ne vibre pas en début de sketch. Si tel est le cas, il y a de fortes chances pour que vous ayez interverti les quatre branchements ;
- que la connexion de masse commune est bien établie entre la carte Arduino et la source de tension externe ;
- que vous n'ayez pas raccordé ensemble les deux pôles de tension d'alimentation de la carte et de la source de tension externe, qui sont marqués d'un +. Sinon, destruction de la carte Arduino assurée !

Qu'avez-vous appris ?

- Vous avez découvert comment commander un moteur pas-à-pas bipolaire.
- La commande a été réalisée au moyen du circuit de commande du moteur L293DNE.
- Vous avez vu comment programmer vous-même une commande de moteur pas-à-pas.

Montage 17

Commande d'un ventilateur

Dans le montage n° 13, vous avez appris à utiliser une thermistance afin d'afficher la valeur mesurée. Dans ce montage, nous verrons comment la fluctuation de la température va déclencher une réaction qui sera non seulement visible, mais aussi perceptible.

Un peu de pratique

Il est temps maintenant de construire quelque chose de bien avec le capteur de température. Que diriez-vous d'ajouter directement plusieurs composants au circuit ? Je pense qu'un ventilateur pour améliorer le climat ambiant et un afficheur pour donner les informations utiles seraient des projets intéressants. Le circuit et le sketch doivent être en mesure de mettre en route un moteur de ventilateur quand une certaine température est atteinte et de l'arrêter quand elle ne l'est plus. Nous touchons ici à l'art et la manière de commander un moteur.

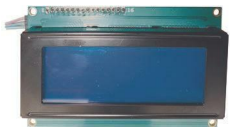
Comme un moteur a assurément besoin de plus de courant et de tension pour fonctionner que la carte Arduino ne peut en fournir, il nous faut trouver autre chose. Cette fois-ci, nous n'utiliserons pas de circuit de commande de moteur. En effet, cette situation de commande d'un seul moteur étant légèrement différente, nous explorerons une autre voie. Je vous exposerai une mesure de sécurité incontournable dans le cadre de la commande d'un composant qui possède une bobine. C'est notamment le cas du moteur ou du relais. Le circuit de commande L293 se termine par les lettres *DNE,* ce qui signifie que le composant possède un circuit de protection qui bloque l'inductance, c'est-à-dire les courants induits très forts lors de la commutation des bobines. Nous y reviendrons bientôt.

Voyons tout d'abord les composants nécessaires pour ce montage.

Composants nécessaires

Ce montage nécessite les composants suivants.

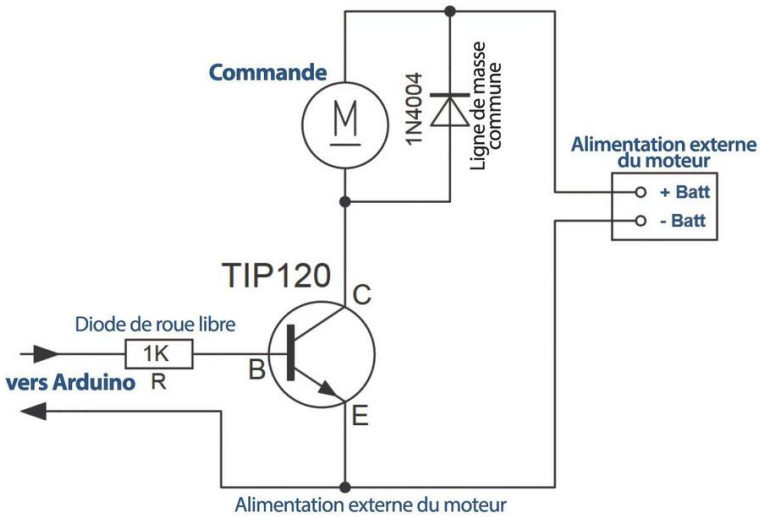
Tableau 17-1 ▶ Liste des composants

Composant	
1 ventilateur de 12 V	
1 transistor de puissance TIP 120	
1 thermistance LM35	
1 diode 1N4004	
2 résistances de 10K	
1 résistance de 1K	
1 module afficheur LCD YwRobot I²C	

Avant de nous intéresser au circuit de commande, nous allons nous attarder sur la problématique évoquée plus haut.

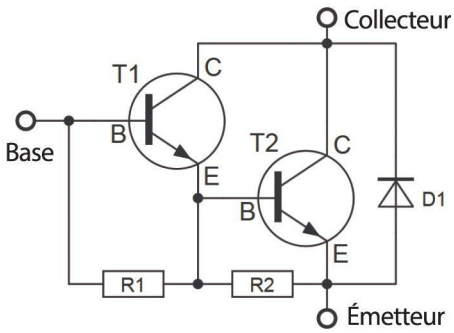
Schéma d'un circuit de commande de moteur

Dans le circuit suivant, un moteur est commandé au moyen d'un transistor Darlington.



◀ Figure 17-1
Circuit de commande d'un moteur avec une diode de roue libre

Le transistor qui est utilisé ici est un peu particulier. J'ai déjà mentionné qu'il s'agit d'un transistor Darlington – ici de type TIP 120. Il se compose effectivement de deux transistors qui sont montés l'un à la suite de l'autre et qui ont donc pour effet de produire une plus forte amplification en courant. Par conséquent, il sera utilisé en présence d'une charge élevée lorsque la tension de commande ne doit pas être sollicitée. La figure 17-2 représente le schéma de connexion d'un transistor de type NPN.



◀ Figure 17-2
Schéma de connexion d'un transistor de type NPN

Revenons-en au circuit de commande du moteur qui, parallèlement à ses raccordements, possède une diode de roue libre. À quoi cela sert-il ? Pour qu'un moteur puisse tourner, il doit notamment comporter une bobine, ou inductance, qui crée un champ magnétique. Cette inductance est dotée d'une propriété spéciale qui fait qu'elle s'efforce de corriger une modification voulue. Quand une tension est appliquée, un courant circule plutôt à contresens à l'intérieur du très long fil enroulé de la bobine, ce qui crée un

champ magnétique. Ce dernier a non seulement pour effet de faire tourner l'axe du moteur, mais il induit à son tour une tension à l'intérieur de la bobine. Ce processus est appelé auto-induction. La bobine se rebelle en quelque sorte, car la tension induite est orientée de manière à s'opposer au courant qui l'a provoquée, ce qui fait que la tension ne s'accumule que lentement dans la bobine. Si par contre je coupe à nouveau le courant, la variation rapide du champ magnétique génère une tension induite qui s'oppose à la baisse de tension et s'avère plusieurs fois plus élevée que la tension initiale. La bobine est donc condamnée à juguler l'afflux de courant en se servant de l'énergie emmagasinée dans le champ magnétique.

C'est là tout le problème. La commutation avec le léger retard ne constitue pas un risque pour le circuit et ses composants. Lors de l'arrêt en revanche, l'effet secondaire extrêmement néfaste de la pointe de tension excessive (> 100 V) doit être impérativement évité pour que le circuit ne grille pas. Faute de quoi, les chances de survie du transistor sont réellement minces. Aussi une diode est-elle connectée en parallèle au moteur pour écrêter la pointe de tension et dériver le courant vers la source. La figure suivante illustre les deux scénarios possibles.

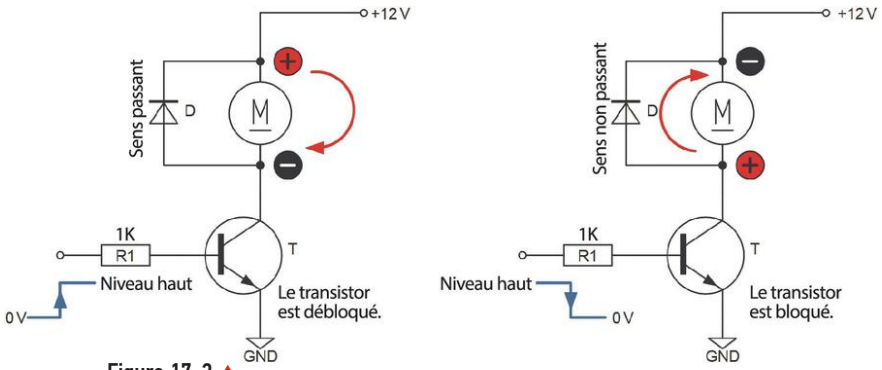
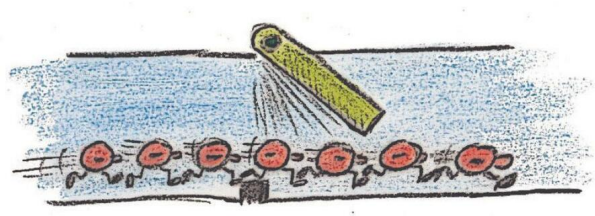


Figure 17-3 ▲
Diode de roue libre en action

Quand le transistor à gauche est débloqué, le moteur démarre, de sorte que les potentiels indiqués se mettent en place au niveau de la diode : le plus à la cathode et le moins à l'anode. Autrement dit, la diode est bloquée et le circuit se comporte comme si elle n'était pas là. Si par contre la base du transistor est reliée à la masse, celui-ci se bloque et les potentiels indiqués apparaissent du fait de la variation du champ magnétique de la bobine : le plus à l'anode et le moins à la cathode. La diode travaille dans le sens de la conduction et dérive le courant vers l'alimentation. Le transistor est préservé. Maintenant que vous savez tout cela, nous allons pouvoir nous intéresser à notre circuit.

La diode est donc un composant qui est capable d'influencer le sens de circulation du courant. C'est un composant conçu à base de semi-conducteurs (silicium ou germanium). Elle a la propriété de ne laisser passer le courant que dans un seul sens (sens passant). Dans le sens inverse, le courant qui circule à travers une diode est pratiquement nul. Ce comportement électrique fait penser à une soupape de chambre à air : l'air de la pompe entre, mais aucun air ne ressort.



◀ Figure 17-4
Électrons traversant la diode dans le sens passant

On voit que les électrons n'ont aucun mal à traverser la diode. Le clapet interne s'ouvre et les électrons circulent sans problème. Les suivants n'auront pas cette chance...

◀ Figure 17-5
Électrons tentant de travers la diode dans le sens non passant

Le clapet ne s'ouvre pas dans le sens souhaité, et on se bouscule au checkpoint (ou point de contrôle), car rien ne bouge.

La forme et la couleur des diodes sont des plus variées. Voici deux exemples.

Le sens dans lequel la diode est passante a une énorme importance, et la présence d'un marquage sur le corps du composant est indispensable. Il ne s'agit pas cette fois-ci d'un code couleur, mais d'un trait plus ou moins épais avec une inscription dessus. Les deux polarités de la diode portent également des noms différents :

- l'anode ;
- la cathode.

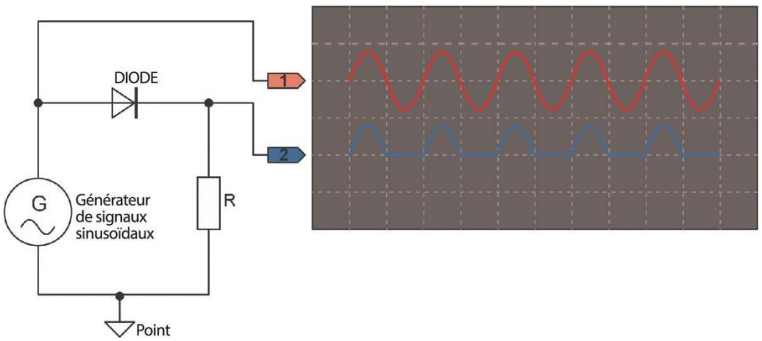
Une diode au silicium est polarisée dans le sens passant quand la différence de tension entre l'anode et la cathode est supérieure à environ +0,7 V.

Figure 17-6 ▶
Symboles de la diode, version en contour à gauche et version pleine à droite

Mais où se situe l'anode par rapport à la cathode ? Voilà le moyen mnémotechnique que j'ai trouvé pour m'en souvenir : le mot « cathode » commence en allemand par la lettre K *(Kathode)*, qui possède un trait vertical sur la gauche. On retrouve ce trait vertical sur le symbole de la diode, à droite, où figure donc la cathode. Physiquement, cette dernière se repère par l'anneau inscrit directement sur le corps de la diode.

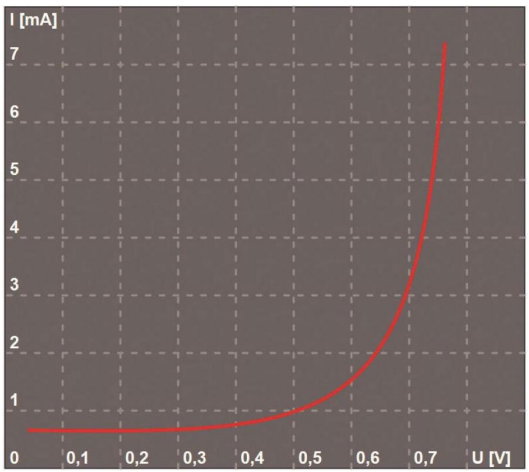
Facile à se rappeler, non ? Voyons maintenant un peu comment fonctionne la diode dans un circuit. J'utilise à l'entrée de cette dernière non pas un signal rectangulaire, mais un signal sinusoïdal, présentant aussi bien des valeurs de tension positives que négatives. Le circuit doit vous être familier maintenant.

Figure 17-7 ▶
Circuit pour commander une diode au moyen d'un générateur de signaux sinusoïdaux



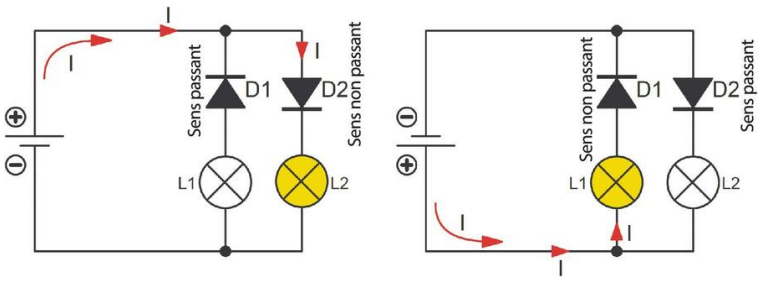
L'entrée de la diode, c'est-à-dire l'anode, est raccordée à la sortie du générateur de signaux sinusoïdaux. Ce point est représenté par une courbe rouge dans l'oscillogramme. La sortie, c'est-à-dire la cathode, est représentée par la courbe bleue. Le signal d'entrée rouge forme une belle courbe sinusoïdale. La diode au silicium ne laissant cependant passer que des signaux positifs > +0,7 V bloque les signaux négatifs. La courbe de sortie (en bleu) ne représente que l'alternance positive de la sinusoïde. Lorsque la tension à l'entrée est négative, la tension de sortie est nulle (diode bloquée).

Avant d'en finir avec la diode, jetons un coup d'œil sur la caractéristique tension-courant. Cette courbe montre à partir de quelle tension d'entrée le courant se met à traverser la diode, et la diode à être conductrice. On ne détecte la présence d'un courant de conduction qu'à partir de +0,5 V environ, et qui augmente très rapidement à partir de +0,7 V.



◀ Figure 17-8
Courbe caractéristique tension-courant d'une diode au silicium

Les deux circuits très simples ci-après montrent le mode de fonctionnement décrit à l'instant pour une vanne électronique. Ils se composent de deux diodes et de deux lampes alimentées par une pile.



◀ Figure 17-9
Sens passant et non passant de diodes dans deux circuits à lampes

Circuit de gauche

Le pôle positif de la pile est relié à l'anode de la diode D2, qui est polarisée dans le sens passant et laisse passer le courant. La lampe L2 s'allume. La diode D1 est bloquée, car sa cathode est reliée au pôle positif de la pile. La lampe L1 reste éteinte.

Circuit de droite

La polarité de la pile est permutée et le pôle positif se trouve en bas ; les rapports de polarité sont inversés. Le pôle positif de la pile est appliqué à l'anode de la diode D1 et la lampe L1 s'allume. La diode D2 est bloquée, car le pôle positif se trouve connecté à sa cathode. La lampe L2 reste éteinte.

Vous vous demandez peut-être maintenant à quoi servent de tels composants. Les domaines d'application sont multiples. En voici quelques-uns :

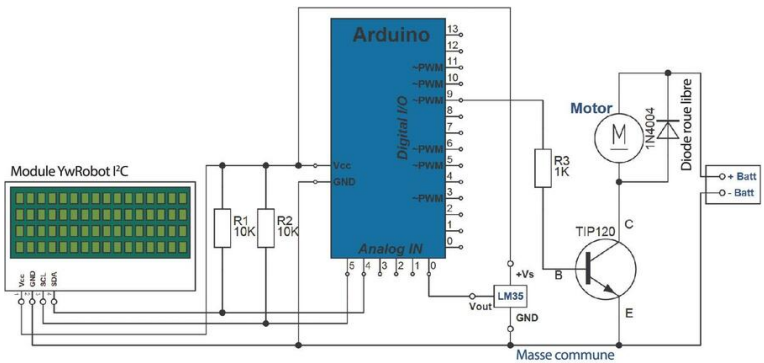
- redressement de courant alternatif ;
- stabilisation de tension ;
- diode de roue libre (protection contre la surtension aux bornes d'une inductance lors de l'arrêt, par exemple d'un moteur), comme dans ce montage ;

Il existe de nombreux types de diodes, par exemple des Zener ou à effet tunnel. Nous ne pouvons toutes les énumérer ici et expliquer leurs différences. Je vous renvoie par conséquent à la littérature spécialisée ou à Internet.

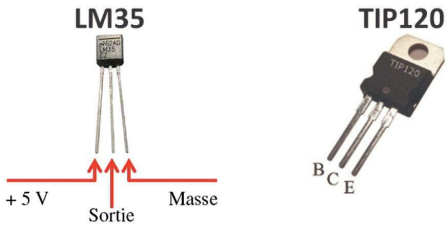
Schéma d'un circuit de commande du ventilateur

Nous avons à gauche le LCD I²C avec les résistances pull-up de 10K, au centre notre Arduino et à droite le capteur de température LM35. Complètement à droite se trouve la commande du moteur avec le transistor TIP 120 et la diode de roue libre 1N4004.

Figure 17-10 ▶
Circuit de commande du ventilateur



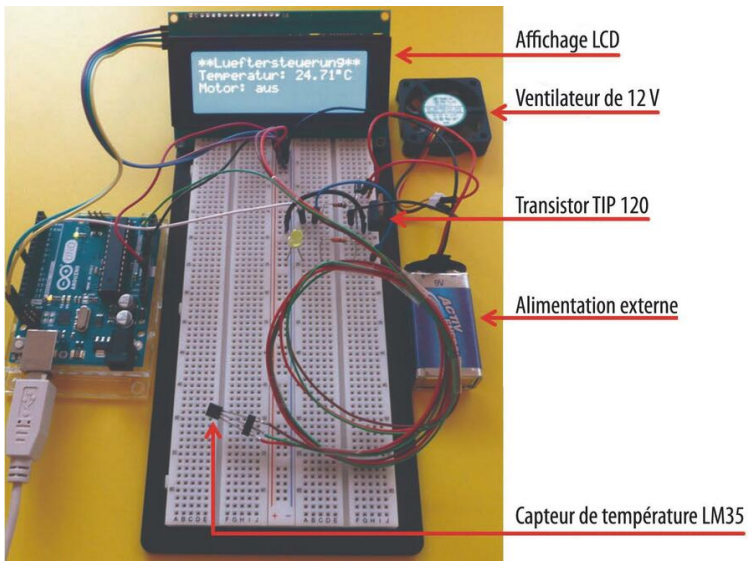
Pour éviter toute confusion, voici le brochage de la thermistance LM35, ainsi que celui du transistor Darlington TIP 120 :



◀ **Figure 17-11**
Brochage de la thermistance *LM35* et du transistor Darlington *TIP 120*

Réalisation du circuit

La réalisation du circuit est rapide sur une petite plaque d'essais.



◀ **Figure 17-12**
Réalisation complète du circuit de commande du ventilateur

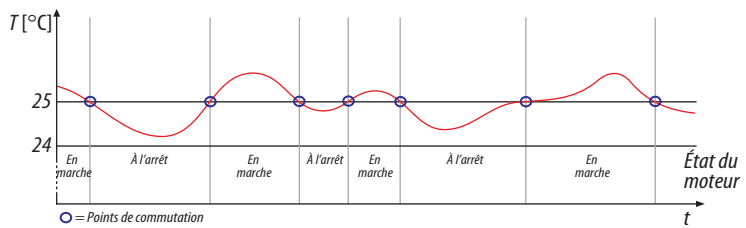
Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Revue de code

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#define sensorPin 0    // Connexion de la sortie du LM35
#define DELAY1 10      // Bref temps d'attente lors de la mesure
#define DELAY2 500     // Bref temps d'attente lors de l'affichage
#define motorPin 9     // Broche de commande du ventilateur
#define threshold 25   // Température de commutation du ventilateur
                       // (25 degrés Celsius)
#define hysterese 0.5  // Valeur d'hystérésis (0,5 degré Celsius)
const int cycles = 20; // Nombre de mesures
// Adresse I2C sur 0x27
LiquidCrystal_I2C lcd(0x27, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);

void setup() {
  pinMode(motorPin, OUTPUT);
  lcd.begin(20,4); // Initialisation du LCD
  lcd.backlight(); // Activation du rétroéclairage
}

void loop() {
  float resultTemp = 0.0;
  for(int i = 0; i < cycles; i++) {
    int analogValue = analogRead(sensorPin);
    float temperature = (5.0 * 100.0 * analogValue) / 1024;
    resultTemp += temperature; // Addition des valeurs mesurées
    delay(DELAY1);
  }
  resultTemp /= cycles;       // Calcul de la moyenne
  lcd.clear();                // Effacer écran LCD
  lcd.setCursor(0,0);         // Positionnement curseur sur 1re ligne
  lcd.print("**Commande ventilo**");
  lcd.setCursor(0,1);         // Positionnement curseur sur 2e ligne
  lcd.print("Temperature : ");
  lcd.print(resultTemp);
  lcd.write(0xD0 + 15);       // Caractère degré (Arduino 1.00)
  lcd.print("C");
  lcd.setCursor(0,2);         // Positionnement curseur sur 3e ligne
  lcd.print("Moteur : ");
  if(resultTemp > (threshold + hysterese))
    digitalWrite(motorPin, HIGH);
  else if(resultTemp < (threshold - hysterese))
    digitalWrite(motorPin, LOW);
  lcd.print(digitalRead(motorPin) == HIGH?"en marche":"stop");
  delay(DELAY2);
}
```

La détermination de la température est effectuée de la même manière et se trouve être la même que dans le montage n° 13. Vous pouvez y lire sans problème la température et l'état du moteur :

Imaginez la situation suivante : le ventilateur doit, tout comme dans notre exemple, se mettre en marche à 25 °C et apporter un peu d'air frais à ceux qui transpirent sur leur Arduino. La température ambiante n'étant cependant pas constante à 100 %, le capteur est lui aussi soumis à certaines fluctuations. Un état est donc par exemple atteint, dans lequel la température mesurée varie constamment entre 24,8 et 25,2 °C. Autrement dit, le ventilateur n'arrête pas de s'allumer et de s'éteindre. Plutôt énervant à la longue ! Voyons maintenant la figure suivante de plus près.



◀ Figure 17-13
En cas de température fluctuant autour de la valeur de consigne, l'état du moteur change constamment.

C'est là que l'*hystérésis* (le mot vient du grec et signifie retard) entre en jeu. On peut expliquer ainsi le comportement d'une régulation avec hystérésis : la variable de sortie, qui commande ici le moteur, ne dépend pas seulement de la variable d'entrée délivrée par le capteur. L'état de la variable de sortie, qui régnait auparavant, joue aussi un rôle important. Dans notre exemple, nous avons une valeur-seuil de 25 °C et une hystérésis de 0,5 °C. Voyons maintenant de plus près la régulation du ventilateur :

```
if(resultTemp > (threshold + hysterese))
   digitalWrite(motorPin, HIGH);
else if(resultTemp < (threshold - hysterese))
   digitalWrite(motorPin, LOW);
```

Quand le ventilateur se met-il en marche ?

Si la condition :

```
resultTemp > (threshold + hysterese) ...
```

est remplie, le ventilateur commence à tourner. C'est le cas ici quand la température mesurée est supérieure à 25 + 0,5 °C.

Quand le ventilateur s'arrête-t-il ?

Si la condition :

```
resultTemp < (threshold - hysterese)
```

est remplie, le ventilateur s'arrête de tourner, ici en l'occurrence quand la température est inférieure à 25 – 0,5 °C. Pour résumer :

- le ventilateur est en marche si : température > 25,5 °C ;
- le ventilateur est à l'arrêt si : température < 24,5 °C.

Voyons maintenant la figure suivante de plus près.

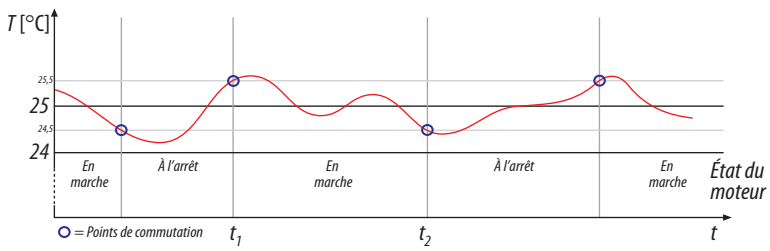


Figure 17-14 ▶ En cas de fluctuation de la température autour de la valeur de consigne, l'état du moteur ne change pas constamment.

Si vous regardez la courbe entre les points t_1 et t_2, vous verrez que la température passe constamment au-dessus et en dessous des 25 °C. Sans commande avec hystérésis, nous aurions sans cesse moteur en marche et moteur à l'arrêt.

Exercice complémentaire

Il existe bien entendu beaucoup d'autres capteurs de température. En voici une sélection :

- TMP75 (avec bus I²C) ;
- AD22100 (capteur de température analogique) ;

- DHT11 (capteur de température et humidité avec microcontrôleur 8 bits intégré) ;
- DS1820 (capteur de température numérique 1-Wire).

Essayez de construire un circuit avec ces capteurs, sans oublier d'adapter le sketch en conséquence. L'afficheur à 4 lignes pourrait aussi très bien servir à montrer l'humidité ambiante, par exemple. Ainsi, vous réaliserez une petite station météo.

Problèmes courants

Si le ventilateur ne se met pas en marche alors que la température-seuil plus la valeur d'hystérésis est atteinte, éteignez tout et vérifiez ce qui suit.

- Le câblage est-il correct ?
- Pas de court-circuit éventuel ?
- La masse commune à la carte Arduino et à la source de tension externe est-elle bien établie ?
- La diode de roue libre est-elle montée dans le bon sens ?
- Si on ne voit rien sur le LCD, le contraste n'est-il pas trop faible ?

Qu'avez-vous appris ?

- Vous avez notamment appris pourquoi une diode de roue libre est indispensable en cas de commande d'une bobine.
- Vous avez appris à utiliser un affichage YwRobot commandé via le bus I^2C pour présenter la valeur de température. Vous pouvez aussi utiliser un autre modèle d'écran LCD qui est commandé par le bus I^2C et pris en charge par Arduino. Vous trouverez des informations plus précises sur Internet. N'oubliez pas de vérifier que l'adresse I^2C du module est correcte.
- Pour que le ventilateur fonctionne correctement, vous avez dû recourir à une alimentation externe, elle-même connectée au transistor de puissance TIP 120. Nous en avons profité pour étudier le fonctionnement du transistor Darlington et vous en présenter le rôle.
- Vous avez appris comment une diode 1N4004 sert, en tant que diode de roue libre, à protéger votre carte Arduino.

Montage

18

Faire de la musique avec Arduino

Comme le titre de ce montage le laisse deviner, il s'agit ici de son, et plus précisément de création de sons. Vous allez connecter un élément piézoélectrique à votre carte Arduino afin de produire des sons. Nous en profiterons également pour vous présenter un jeu qui stimulera vos cellules grises.

Y a pas le son ?

À la longue, vous en avez peut-être assez des signaux lumineux et autres LED clignotantes. Aussi allons-nous voir à présent comment votre carte Arduino peut émettre des sons au moyen d'un *élément piézoélectrique*.

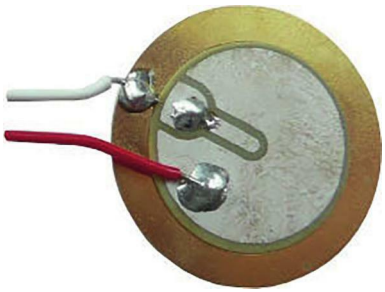


Figure 18-1
Disque piézo

Vous ne risquez pas les ondes de choc acoustiques avec un piézo, car les vibrations émises couvrent un espace des plus réduits. Il est cependant parfait pour ce que nous allons faire. Sa forme est assez bizarre et on a du mal à croire que ce composant puisse faire du bruit. Il renferme un cristal qui se met à vibrer quand une tension alternative lui est appliquée. Cet effet appelé piézoélectrique se produit quand des forces (pression ou déformation) sont exercées sur certains matériaux ;

une tension électrique est alors mesurable. Le buzzer piézoélectrique agit de façon inverse : quand une tension alternative est appliquée, une déformation régulière, perçue comme une vibration, se produit et met en mouvement les molécules d'air, ce qui est perçu comme un son. Pour qu'il soit plus fort, le mieux est de coller le buzzer piézoélectrique sur un support vibrant, de sorte que les vibrations émises soient transmises et amplifiées. Voici comment peut être représenté un buzzer piézoélectrique :

Figure 18-2 ▶
Symbole du buzzer piézoélectrique

Si on branche, par exemple, l'élément sur une sortie numérique et si on passe à intervalles réguliers la sortie en niveau `HIGH` ou `LOW`, on entend un craquement dans l'élément piézoélectrique. Plus le laps de temps entre niveaux `HIGH` et `LOW` est court, plus le son audible est aigu ; plus le laps de temps est long, plus le son est grave. Le phénomène est le même quand, par exemple, vous passez vos doigts plus ou moins vite sur une grille à lamelles. Plus vous allez vite, plus le bruit est aigu ; le *piézo* fonctionne sur ce principe. Un craquement répété, tantôt plus lent, tantôt plus rapide, influe sur la fréquence du son.

Composant nécessaire

La première partie de ce montage nécessite le composant suivant. Par la suite, nous aurons besoin d'autres composants que nous vous préciserons le moment venu.

Tableau 18-1 ▶
Liste des composants

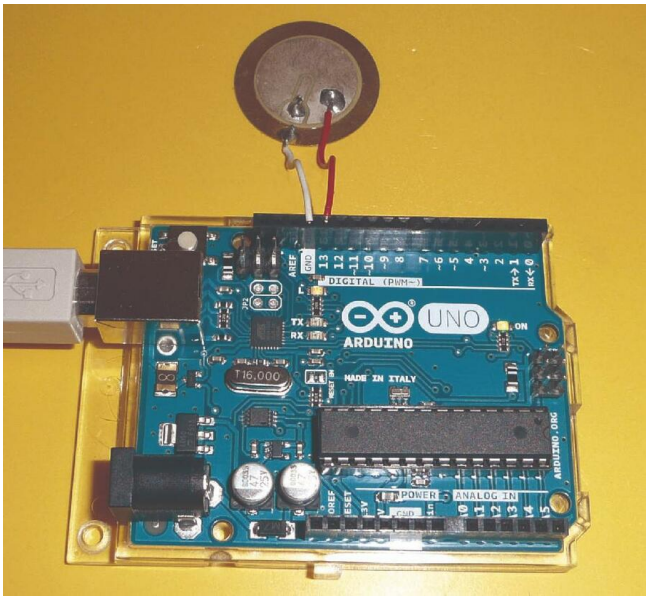
Composant	
Élément piézoélectrique	

Schéma

Le schéma est très simple, car l'élément piézoélectrique doit être raccordé à une broche numérique et à la masse de la carte Arduino, sans l'aide de composants supplémentaires. Je n'en inclus donc pas ici.

Réalisation du circuit

Le circuit est aussi très simple puisque je me suis contenté de raccorder l'élément piézoélectrique par des fils aux broches adéquates.



◀ **Figure 18-3**
Réalisation du circuit de commande de l'élément piézoélectrique

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Revue de code

Le sketch suivant active et désactive la broche numérique à laquelle l'élément piézoélectrique est connecté. Une pause est ménagée entre ces deux actions.

```
#define piezoPin 13 // Élément piézoélectrique sur broche 13
#define DELAY 1000  // Valeur de la pause

void setup() {
  pinMode(piezoPin, OUTPUT);
}

void loop() {
  digitalWrite(piezoPin, HIGH); delayMicroseconds(DELAY);
  digitalWrite(piezoPin, LOW);  delayMicroseconds(DELAY);
}
```

Ne vous inquiétez pas pour la fonction `delayMicroseconds`. Son action est la même que celle de la fonction `delay`, à ceci près que la valeur transmise n'est pas interprétée en millisecondes, mais en microsecondes ; la microseconde est 1 000 fois plus petite (1 ms = 1 000 µs). Cette nouvelle fonction est utilisée, car `delay` ne permet pas de descendre en dessous de 1 ms. Pour le premier sketch qui doit être capable d'émettre plusieurs sons à des fréquences différentes, mieux vaut créer un tableau des sons avec différentes valeurs que nous appellerons l'une après l'autre pendant le sketch. On utilise pour ce faire la fonction `tone` (sons) mise à disposition par Arduino.

```
#define piezoPin 13      // Élément piézoélectrique sur broche 13
#define toneDuration 500 // Durée du son
#define tonePause 800    // Longueur de la pause entre les sons
int tones[] = {523, 659, 587, 698, 659, 784, 698, 880};
int elements = sizeof(tones) / sizeof(tones[0]);

void setup() {
  noTone(piezoPin);                          // Rendre le piézo muet
  for(int i = 0; i < elements; i++) {
    tone(piezoPin, tones[i], toneDuration); // Exécuter le son
    delay(tonePause);                        // Pause entre les sons
  }
}

void loop(){ /* vide*/ }
```

Le tableau unidimensionnel `tones` est du type de donnée `int` et contient les fréquences en *hertz* des sons à exécuter. Les hertz (Hz) servent à mesurer le nombre de vibrations par seconde. Plus la valeur est élevée, plus le son est aigu, et vice versa. Le nombre d'éléments du tableau est affecté à la variable `elements` ; il servira plus tard dans la boucle `for` pour traiter tous les éléments. Le réglage manuel de la limite supérieure ou de la condition de la boucle `for` est ainsi évité, celui-ci étant fait automatiquement au moyen d'un calcul.

On utilise pour ce faire la fonction `sizeof` de C++, qui détermine la taille d'une variable ou d'un objet dans la mémoire. Voici un court exemple :

```
byte byteValue = 16;          // Variable du type byte
int intValue = 4;             // Variable du type int
long longValue = 3.14;        // Variable du type long
int myArray[] = {25, 46, 9};  // Variable du type int

void setup() {
  Serial.begin(9600);
  Serial.print("Nombre d'octets pour 'byte' : ");
  Serial.println(sizeof(byteValue));
  Serial.print("Nombre d'octets pour 'int':  ");
```

```
  Serial.println(sizeof(intValue));
  Serial.print("Nombre d'octets pour 'long': ");
  Serial.println(sizeof(longValue));
  Serial.print("Nombre d'octets pour 'myArray': ");
  Serial.println(sizeof(myArray));
}

void loop() { /* vide */ }
```

L'affichage est donc le suivant :

```
Nombre d'octets pour 'byte' : 1
Nombre d'octets pour 'int' :  2
Nombre d'octets pour 'long' : 4
Nombre d'octets pour 'myArray' : 6
```

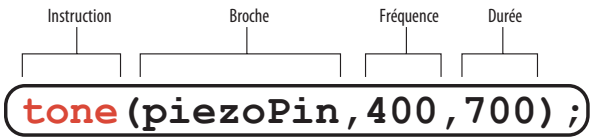
Quand on regarde les valeurs pour les types de données `byte`, `int` et `long`, on s'aperçoit qu'elles sont identiques à celles indiquées dans le chapitre 3, dans lequel il était question de types de données et de domaines de valeurs. Passons à la dernière ligne de l'affichage. On y voit que le tableau occupe 6 octets de mémoire, ce qui est logique puisqu'un seul élément `int` nécessite 2 octets de mémoire. Or, nous avons 3 éléments. Le résultat est donc $2 \times 3 = 6$ octets. La ligne

```
int elements = sizeof(tones) / sizeof(tones[0]);
```

divise le nombre d'octets du tableau par le nombre d'octets d'un seul élément. C'est toujours de cette manière qu'on obtient le nombre d'éléments d'un tableau. Mais revenons à notre sketch. Tout au début, la fonction `noTone` rend le piézo muet au cas où il devrait encore pépier du fait d'un sketch précédent. Elle n'a qu'un seul paramètre, qui indique la broche sur laquelle se trouve le piézo.

◀ Figure 18-4
La fonction `noTone` rend le piézo muet.

La fonction `tone` possède en revanche deux autres paramètres. L'un indique la *fréquence* et l'autre la *durée* pendant laquelle le son doit être audible.



◀ Figure 18-5
La fonction `tone` rend le piézo bavard.

Peut-être vous demandez-vous comment nous en sommes venus aux différentes valeurs utilisées dans le tableau des sons. Non, nous ne les avons pas toutes testées pour savoir lesquelles convenaient ! Elles sont tirées d'un exemple de sketch qui se trouve dans l'IDE Arduino. Recherchez le fichier `pitches.h` dans le dossier `C:\Program Files (x86)\Arduino\examples\02.Digital\toneMelody`. Vous y trouverez les fréquences correspondant à de nombreuses notes. Vous pouvez inclure ce fichier dans votre sketch et utiliser ensuite directement les constantes symboliques. Essayez ! Le code est alors beaucoup plus parlant et plus clair que lorsque des valeurs numériques sont utilisées. Nous allons maintenant essayer de faire quelque chose d'utile en nous intéressant à un jeu mêlant sons et couleurs. Comme il nécessite la construction d'un shield, il sera d'autant plus intéressant.

Jeu de la séquence des couleurs

Vous allez mettre en pratique ce que vous venez d'apprendre dans un jeu intéressant dit de la séquence des couleurs. Nous avons 4 LED de couleurs différentes disposées en carré. Près de chacune se trouve un bouton-poussoir. Le microcontrôleur imagine un motif pour l'ordre d'allumage des LED ; à vous de le reproduire correctement. Au début, la séquence ne comporte qu'une seule LED allumée ; une nouvelle vient s'ajouter après chaque bonne réponse. L'allumage de chacune des quatre LED est accompagné d'un son qui lui est propre. Le jeu ravit donc non seulement les yeux, mais aussi les oreilles. J'ai ajouté au circuit un shield avec une face avant de ma fabrication. Voyez plutôt la figure 18-6.

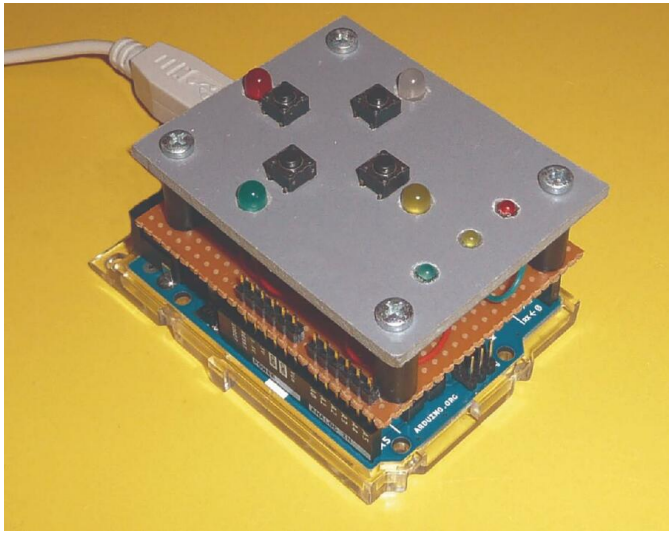
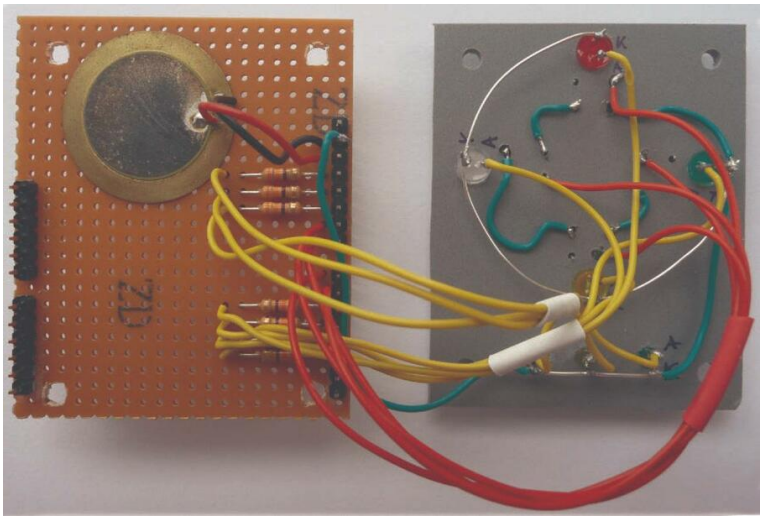


Figure 18-6 ▶
Shield avec la face avant pour le jeu de la séquence des couleurs

Sur la face avant, on voit les 4 grosses LED de 5 mm avec leur bouton-poussoir respectif. Quand une LED s'allume, on doit appuyer sur le bouton-pous-

soir placé à côté. La partie basse de la face avant est occupée par 3 plus petites LED de 3 mm. Elles servent à afficher l'état, sur lequel je reviendrai plus tard. La figure suivante montre bien le shield et la face avant avec le câblage.



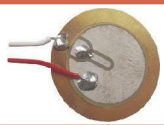
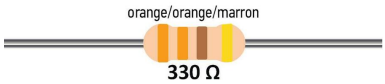
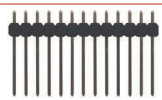
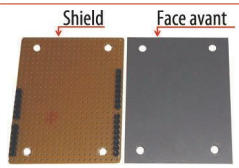
◀ **Figure 18-7**
Shield ouvert et la face avant retournée

Les choses sont moins compliquées qu'elles n'y paraissent, et la construction devient claire quand on regarde le schéma. Voici les points que je poserai comme conditions pour le jeu :

- une longueur déterminée de la séquence, d'abord constante, est fixée par le sketch ;
- chacune des 4 LED doit avoir sa propre note avec une fréquence particulière ;
- quand une des 4 LED s'allume, la note correspondante est émise ;
- quand le bouton-poussoir situé à côté est pressé, la LED s'allume et la note correspondante est émise ;
- si la séquence a été correctement reproduite, la LED d'état verte s'allume et une suite de sons crescendo se fait entendre. Le jeu reprend ensuite au début avec une nouvelle séquence ;
- si la séquence reproduite est fausse à un endroit quelconque, la LED d'état rouge s'allume et une suite de sons decrescendo se fait entendre. Le jeu redémarre ensuite avec une nouvelle séquence.

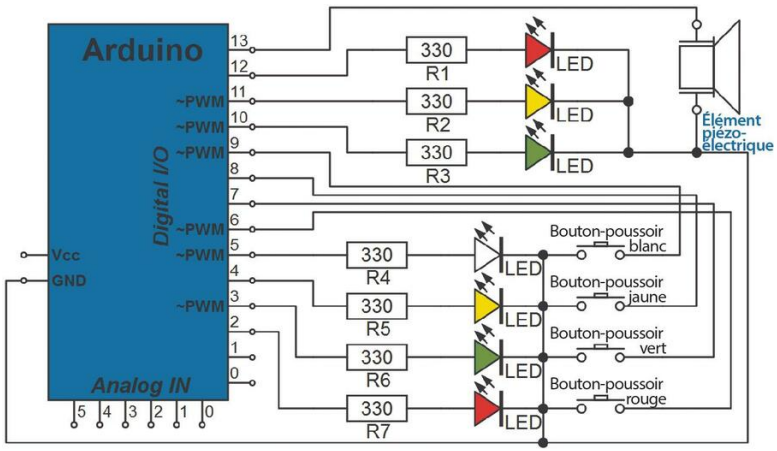
Examinons maintenant les composants nécessaires pour la réalisation du jeu.

Tableau 18-2 ▶
Liste des composants

Composant	
1 élément piézoélectrique	
4 LED (de différentes couleurs)	
3 LED de 3 mm (de différentes couleurs)	
4 boutons-poussoirs miniatures	
7 résistances de 330 Ω	
4 entretoises DK 15 mm, en plastique	
4 vis M3 30 mm coupées à 23 mm environ + 4 écrous	
2 barrettes à 6 broches + 2 barrettes à 8 broches (des barrettes plus longues peuvent être utilisées avec la carte Arduino Uno Rev3)	
1 shield + 1 face avant (la face avant en mousse dure s'achète dans les magasins de bricolage)	

Voyons maintenant le schéma.

Schéma



◀ Figure 18-8
Circuit du jeu de la séquence des couleurs

Sketch du jeu de la séquence des couleurs

Voici maintenant le code du sketch quelque peu élargi. Les explications suivent.

```
#define MAXARRAY              5 // Définir la longueur de la séquence
int ledPin[] = {2, 3, 4, 5};    // Tableau des LED avec numéros de broche
#define piezoPin             13 // Broche piézo
#define buttonPinRed          6 // Broche bouton-poussoir LED rouge
#define buttonPinGreen        7 // Broche bouton-poussoir LED verte
#define buttonPinYellow       8 // Broche bouton-poussoir LED orange
#define buttonPinWhite        9 // Broche bouton-poussoir LED blanche
#define ledStatePinGreen     10 // LED d'état verte
#define ledStatePinYellow    11 // LED d'état orange
#define ledStatePinRed       12 // LED d'état rouge
int colorArray[MAXARRAY];     // Contient la suite de chiffres pour les
                              // couleurs à afficher
int tones[] = {1047, 1175, 1319, 1397}; // Fréquences des sons pour les
                                        // 4 couleurs
int counter = 0;              // Nombres de LED actuellement allumées
boolean fail = false;

void setup() {
  Serial.begin(9600);
  for(int i = 0; i < 4; i++)
     pinMode(ledPin[i], OUTPUT); // Programmation des broches de LED
                                 // comme sortie
  pinMode(buttonPinRed,     INPUT_PULLUP);
  pinMode(buttonPinGreen,   INPUT_PULLUP);
  pinMode(buttonPinYellow,    INPUT_PULLUP);
  pinMode(buttonPinWhite,   INPUT_PULLUP);
```

```
  pinMode(ledStatePinGreen, OUTPUT);
  pinMode(ledStatePinYellow,  OUTPUT);
  pinMode(ledStatePinRed,   OUTPUT);
}

void loop() {
  Serial.println("Départ du jeu");
  generateColors();
  int buttonCode;
  for(int i = 0; i <= counter; i++) {    // Boucle extérieure
    giveSignalSequence(i);
    for(int k = 0; k <= i; k++) {       // Boucle intérieure
      while(digitalRead(buttonPinRed)  && digitalRead(buttonPinGreen) &&
           digitalRead(buttonPinYellow) &&
digitalRead(buttonPinWhite));
      Serial.println("Bouton poussé !"); // Pour contrôle dans moniteur
                                         // série
      if(!digitalRead(buttonPinRed))
        buttonCode = 0;
      if(!digitalRead(buttonPinGreen))
        buttonCode = 1;
      if(!digitalRead(buttonPinYellow))
        buttonCode = 2;
      if(!digitalRead(buttonPinWhite))
        buttonCode = 3;
      giveSignal(buttonCode);
      // Vérifier si la bonne couleur a été pressée
      if(colorArray[k] != buttonCode){
        fail = true;
        break;  // Quitter la boucle for interne
      }
    }

    if(!fail)
      Serial.println("correct"); // Pour contrôle dans moniteur série
    else {
      digitalWrite(ledStatePinRed, HIGH);
      for(int i = 3000; i > 500; i-=150){
       tone(piezoPin, i, 10); delay(20);
      }
      Serial.println("faux");    // Pour contrôle dans moniteur série
      delay(2000);
      digitalWrite(ledStatePinRed, LOW);
      counter = 0; fail = false;
      break;                  // Quitter la boucle for
    }
    delay(2000);

   if(counter + 1 ==  MAXARRAY) {
     digitalWrite(ledStatePinGreen, HIGH);
     for(int i = 500; i < 3000; i+=150){
       tone(piezoPin, i, 10); delay(20);
     }
```

```
      Serial.println("Fini !");   // Pour contrôle dans moniteur série
      delay(2000);
      digitalWrite(ledStatePinGreen, LOW);
      counter = 0; fail = false;
      break; // Quitter la boucle for extérieure
    }
     counter++; // Incrémenter le compteur
   }
}

void giveSignalSequence(int value) {
  // Affichage LED
  for(int i = 0; i <= value; i++){
    digitalWrite(2 + colorArray[i], HIGH);
    generateTone(colorArray[i]); delay(1000);
    digitalWrite(2 + colorArray[i], LOW); delay(1000);
  }
}

void generateTone(int value) {
  tone(piezoPin, tones[value], 1000);
}

void giveSignal(int value) {
  // Affichage LED + signal sonore
  digitalWrite(2 + value, HIGH); generateTone(value); delay(200);
  digitalWrite(2 + value, LOW); delay(200);
}

void generateColors() {
  randomSeed(analogRead(0));
  for(int i = 0; i < MAXARRAY; i++)
    colorArray[i] = random(4);    // Générer des chiffres aléatoires de
                                  // 0 à 3
    // 0 = rouge, 1 = vert, 2 = orange, 3 = blanc
  for(int i = 0; i < MAXARRAY; i++)
    Serial.println(colorArray[i]); // Pour contrôle dans moniteur série
}
```

Revue de code

Une valeur numérique est affectée à chaque couleur à afficher : `0` pour rouge, `1` pour vert, `2` pour orange et `3` pour blanc. Un tableau peut ainsi être initialisé avec des valeurs allant de 0 à 3 ; il pourra ensuite servir à afficher les LED. Supposons que vous ayez un tableau avec les valeurs `0`, `2`, `2`, `1` et `3`, les diodes s'allument donc dans l'ordre suivant : rouge, orange, orange, vert et blanc. Dans notre sketch, son nom est `colorArray` et il reçoit ses valeurs via la fonction `generateColors`. Pour les rendre visibles, la fonction `giveSignal` convertit les valeurs en signaux pour commander les LED.

```
void giveSignalSequence(int value) {
  // Affichage LED
  for(int i = 0; i <= value; i++){
    digitalWrite(2 + colorArray[i], HIGH);
    generateTone(colorArray[i]); delay(1000);
    digitalWrite(2 + colorArray[i], LOW); delay(1000);
  }
}
```

Si la fonction doit toujours afficher la séquence des couleurs, peut-être vous demandez-vous pourquoi nous avons encore besoin d'une variable. Et ce que signifie le `2` qui est utilisé dans la fonction `digitalWrite`.

En fait, la séquence complète ne doit pas s'afficher au début, mais seulement au fur et à mesure avec, chaque fois, une couleur en plus. Le tableau des couleurs `colorArray` contient bien la séquence complète, mais la variable transmise dans `value` indique à la fonction combien d'éléments du tableau doivent être interrogés et affichés. Les 4 grandes LED étant cependant raccordées aux sorties numériques des broches 2 à 5, le chiffre 2 est quasiment un décalage qui indique la broche de démarrage quand les valeurs `0` à `3` sont ajoutées au tableau des couleurs. Bien entendu, il ne faut pas utiliser de *magic numbers*. Vous pouvez naturellement employer une constante symbolique, par exemple avec le nom `COLORPINOFFSET`.

Avant de passer à l'explication de la logique dans la fonction `loop`, nous devons revenir sur la fonction `setup`. Il y a, par exemple, des broches de bouton-poussoir qui sont, bien sûr, programmées comme entrées. Pourtant, quelque chose est envoyé à ces mêmes entrées par la fonction `digitalWrite`. Pourquoi cela ?

J'utilise la possibilité d'activer les résistances pull-up présentes et connectées en interne dans le microcontrôleur. Plus besoin ainsi de connecter des résistances pull-up ou pull-down externes. J'ai déjà expliqué cela dans le montage n° 3. Ça vous dit quelque chose ? Si vous avez oublié, relisez-le !

Mais comment se fait-il que la fonction `loop` s'exécute continuellement ? La première boucle `for`, que nous avons qualifiée de *boucle extérieure*, devrait elle aussi s'exécuter continuellement. C'est pourtant elle qui est chargée d'afficher la séquence en fonction de la variable `counter`.

Oui, bien vu ! La fonction `loop`, qui est une boucle sans fin, devrait normalement s'exécuter en permanence. Seulement, j'ai incorporé un arrêt qui la bloque tant qu'un des quatre boutons-poussoirs n'est pas pressé. Voici la partie de code en question :

```
while(digitalRead(buttonPinRed)  && digitalRead(buttonPinGreen) &&
     digitalRead(buttonPinYellow) && digitalRead(buttonPinWhite));
```

Les entrées numériques, auxquelles les boutons-poussoirs sont raccordés, étant reliées au +5 V à travers les résistances pull-up internes, mon interrogation doit porter sur le niveau `LOW`. Tant que toutes les entrées sont sur niveau `HIGH`, la boucle `while` exécute l'instruction qui vient aussitôt après.

Mais quelle instruction est exécutée au juste ? D'après le code, la ligne `Serial.println("Bouton poussé !");` devrait être exécutée. Mais ça n'a pas beaucoup de sens !

Vous avez raison, ça n'a pas beaucoup de sens ! Vous avez cependant oublié une petite chose. L'instruction, qui vient immédiatement après la boucle `while`, est le point-virgule situé tout à la fin. Il s'agit quasiment d'une instruction vide, qui fait en sorte que la boucle `while`, quand aucun des boutons-poussoirs n'est enfoncé, devienne elle-même une boucle sans fin. C'est une manière élégante d'interrompre ici le déroulement du programme. Ce n'est que quand l'un ou l'autre des boutons est pressé que la condition dans la boucle `while` n'est plus remplie et que le programme reprend son cours. Le bouton pressé est alors identifié afin de comparer la valeur de la couleur concernée à l'élément du tableau qui vient d'être sélectionné dans la boucle intérieure. Si une concordance a été trouvée, on passe à la valeur de couleur suivante dans la séquence. En revanche, si une erreur a été commise, la variable `fail` reçoit la valeur `true`, et l'instruction `break` fait sortir prématurément de la boucle intérieure. Autrement dit, l'instruction `if` :

```
if(!fail)...
```

reprend le cours du programme en conséquence. La variable `counter` est augmentée de la valeur `1`, dans la mesure où aucune erreur n'a été commise et où la fin de la séquence n'est pas encore atteinte, si bien que la prochaine séquence affichée sera plus longue. J'ai laissé les affichages sur le moniteur série dans le code pour une meilleure compréhension des procédés. Ils vous donnent au début la séquence qui a été sélectionnée pour que vous puissiez faire, le cas échéant, un peu d'expérimentation. Lisez une fois le code de bout en bout et essayez de le comprendre.

Problèmes courants

Si aucune des 4 grandes LED ne s'allume ou si le piézo n'émet aucun son une fois le sketch transmis, vérifiez :

- que le câblage est correct ;
- qu'il n'y a pas de court-circuit ;
- que des soudures ne se touchent pas par inadvertance.

Exercice complémentaire

Élargissez le sketch de telle sorte que la séquence soit allongée au démarrage d'un nouveau jeu après chaque reproduction correcte. Vous pouvez également jouer sur les pauses entre les différentes couleurs. Réduisez-les afin que le jeu devienne peu à peu plus difficile. Une des 3 petites LED de 3 mm n'a pas été utilisée dans mon sketch. Essayez donc de lui donner une fonction utile. Vous pouvez, par exemple, la faire s'allumer brièvement quand un nouveau jeu commence. Les possibilités sont à coup sûr nombreuses !

Qu'avez-vous appris ?

- Vous avez découvert comment commander un élément piézoélectrique en générant une fréquence par l'alternance des niveaux `HIGH` et `LOW` sur la sortie numérique correspondante.
- Il est également possible d'influer sur le piézo avec les fonctions `noTone` et `tone`, de manière à le rendre muet ou le faire vibrer à une fréquence voulue.
- Vous avez vu comment fabriquer vous-même une face avant conviviale avec des moyens très simples afin de réaliser un jeu intéressant.

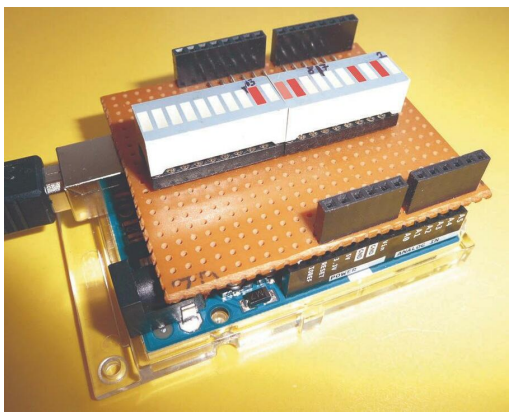
Montage 19

L'Arduino-Talker

Je trouve toujours excitant de pouvoir instaurer une communication entre divers appareils ou différents langages de programmation. C'est parfois un peu compliqué, mais une fois les difficultés surmontées, le plaisir est encore plus grand. Dans ce projet, je voudrais vous montrer comment utiliser l'interface série de votre carte Arduino à l'aide du moniteur série. J'utiliserai d'abord un script Python grâce auquel vous pourrez facilement contrôler toutes les broches. Vous serez ainsi capable de contrôler des broches numériques et analogiques via un signal MLI (voir le montage n° 1). Puis nous étudierons en détail l'application Arduino-Talker qui peut être programmée en langage C#. Le code source complet correspondant figure sur l'extension web de l'ouvrage.

Communiquer avec l'Arduino

Sur l'image ci-dessous, vous pouvez voir le shield de LED que j'ai réalisé. Au lieu d'utiliser des LED individuelles, j'ai employé des afficheurs à barres. Ainsi, les LED sont dans un boîtier compact et parfaitement alignées.



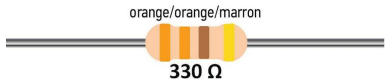
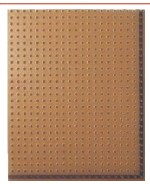
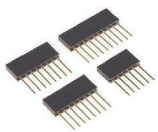
◀ **Figure 19-1**
Le shield d'affichage pour les sorties numériques

Sur cette photo, vous pouvez aussi noter que les LED sont plus ou moins brillantes. Ces LED sont placées sur les broches où un contrôle PWM est possible. Une fonction clignotante est également possible, et si vous regardez assez longtemps l'image, vous verrez peut-être un clignotement ici ou là. Évidemment, vous pouvez construire ce circuit sur une plaque d'essais : l'utilisation d'un shield, comme je l'ai fait ici, n'est pas obligatoire. L'affichage graphique à barres est un composant Kingbright de référence DC-10EWA.

Composants nécessaires

Ce montage nécessite les composants suivants.

Tableau 19-1 ▶
Liste des composants

Composant	
12 LED rouges	
12 résistances de 330 Ω	
1 plaque perforée	
1 Arduino Stackable Header Kit – R3 avec des barrettes empilables : • 2 de 8 broches • 1 de 6 broches • 1 de 10 broches	

Schéma

Le schéma de principe du circuit est si simple que je ne l'ai pas montré à nouveau. Il s'agit d'un contrôle simple de LED avec résistances série correspondantes.

Remarques

Avant de commencer, laissez-moi vous expliquer quel est le but de ce projet et à quoi ressemble le programme de contrôle en langage C#. Comme je l'ai dit au début, nous utiliserons un programme externe pour contrôler les broches numériques du microcontrôleur Arduino. Commençons par

étudier l'interface de l'application Arduino-Talker, également appelée GUI *(Graphical-User-Interface)*.

▲ **Figure 19-2**
L'application Arduino-Talker

Vous reconnaissez plusieurs boutons utilisés pour contrôler les broches. Les options de contrôle sont les suivantes :

- une seule broche On (HIGH) ;
- une seule broche Off (LOW) ;
- clignotement d'une seule broche ;
- contrôle PWM d'une broche ;
- toutes les broches Off (LOW) ;
- toutes les broches On (HIGH).

Ce ne sont que quelques-uns des contrôles que j'ai installés mais bien sûr, il n'y a pas de limites à la créativité. Vous pouvez programmer une fenêtre où certains motifs d'affichage sont mémorisés puis affichés en séquence.

En haut de l'interface de l'application figurent les paramètres de la connexion du port série via lequel nous communiquons avec l'Arduino. Pour que l'Arduino accepte les commandes de l'Arduino-Talker, un sketch spécial est requis, fourni avec l'Arduino Talker. Un clic sur le bouton *Arduino Sketch-Code* affiche ce sketch dans une fenêtre. En bas de l'interface, dans la barre d'état, le statut de la liaison COM est indiqué et la chaîne de caractères envoyée est affichée. Pour que tout fonctionne, j'ai défini un protocole d'échange.

Mais qu'est-ce qu'un protocole ? Quand il y a un échange de données entre ordinateurs ou microcontrôleurs, la communication doit être en quelque sorte réglementée. Si deux personnes parlent, elles doivent déjà utiliser

la même langue afin de se comprendre. La situation est similaire dans le cas des ordinateurs. Un vocabulaire permettant la communication doit être établi pour indiquer à l'Arduino, sous une forme parfaitement claire et compréhensible, les actions à mettre en œuvre. C'est exactement ce qui se passe dans un protocole. Examinons ces deux exemples.

Allumer la LED sur la broche 13

Cliquer sur le bouton correspondant affichera la chaîne générée par le protocole, indiquée à droite et envoyée via l'interface série. La LED sur la broche 13 s'allumera.

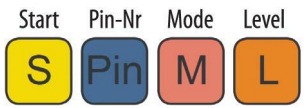
Régler le niveau PWM (MLI) de la LED sur la broche 10

La broche PWM 10 est programmée à la valeur 76, la broche numérique 10 génère le signal PWM correspondant et la LED s'allume donc faiblement. Au début du livre, j'ai déjà expliqué le principe de la modulation PWM ou MLI.

Venons-en maintenant à un exemple concret, car la broche numérique ne sera pas commandée par la fonction `digitalWrite` mais par `analogWrite`. Voyons en détail ce que cela signifie.

Comprendre le code, étape par étape

Le sketch suivant étant assez long, je l'ai divisé en blocs pour le commenter. Je voudrais d'abord revenir sur le protocole que j'ai utilisé. Celui-ci interroge régulièrement le port série pour être en mesure de réagir au flux de données entrant. Tous les flux de caractères n'ont pas d'effet sur les LED connectées. Le protocole de transmission doit le savoir. Pour le tester, vous pouvez entrer directement dans le moniteur série une chaîne appropriée : si le protocole fonctionne bien, vous devrez constater une réaction correspondante de l'Arduino. Veillez aussi à ce que la vitesse de transmission soit correcte, sans quoi rien ne se passera même si le protocole est bon. Voici les détails de ce protocole.



Les blocs individuels ont les significations suivantes :

- S : l'étiquette de début (qui doit être présente, sinon il n'y a pas de contrôle LED) ;
- Pin : le numéro de broche ;
- M : le mode (D pour *digital* : numérique, A pour *analog* : analogique, B pour *blink* : clignotant) ;
- L : le niveau (0/1 : numérique, 0-255 : analogique).

Voici quelques exemples envoyés à la carte Arduino par le moniteur série.

◀ **Figure 19-3**
La broche numérique 03 est réglée sur HIGH.

◀ **Figure 19-4**
La broche numérique 06 est réglée sur le niveau LOW.

◀ **Figure 19-5**
La broche numérique 09 est contrôlée en tant que sortie analogique avec le signal PWM 50.

◀ **Figure 19-6**
La broche numérique 04 est réglée pour clignoter.

Étudions maintenant le sketch.

Définition globale

Les numéros de broches numériques à contrôler sont stockés dans le tableau `pinArray`. Afin de pouvoir faire clignoter les broches, il nous faut aussi un tableau `pinBlinkArray` pour stocker les valeurs booléennes ou l'état clignotant. Les autres variables n'ont pas besoin d'explications supplémentaires. Le contrôle d'intervalle à l'aide de la fonction `millis` a déjà été étudié dans l'un des premiers montages.

```
#define ARRAY_SIZE 12              // Dimension des tableaux des LED
int pinArray[ARRAY_SIZE] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
boolean pinBlinkArray[ARRAY_SIZE]; // Tableau des clignotements
int pin, level;                    // Numéro de broche et niveau
char mode;
                                   // Mode numérique ou analogique
```

```
int interval = 1000;                    // Durée du clignotement
unsigned long prev;                     // Variable de temps
int ledStatus = LOW;                    // État du clignotement
```

Initialisation

Dans la fonction `setup`, la vitesse de transmission de l'interface série est réglée à 38 400 bauds et les broches 2 à 13 du réseau de broches sont définies comme des sorties. Le tableau clignotant est rempli avec des valeurs `false`, donc aucune LED ne clignotera au début.

```
void setup() {
  Serial.begin(38400);
  for(int i = 0; i < ARRAY_SIZE; i++)
    pinMode(pinArray[i], OUTPUT);
  for(int i = 0; i < ARRAY_SIZE; i++)
    pinBlinkArray[i] = false;
}
```

Boucle d'exécution continue

La boucle `loop` appelle maintenant en continu la fonction `readSerial` que nous avons écrite pour récupérer les données série transmises.

```
void loop() {
  readSerial(); // Appel à la fonction d'acquisition
}
```

Récupération des données

Le flux de données de l'interface série est récupéré et évalué via la fonction `readSerial`. Qu'est-elle censée faire? Elle doit contrôler les broches numériques en fonction de certains paramètres. Tout d'abord, nous avons défini le protocole pour que le contrôle commence toujours par le caractère S. Si le flux envoyé n'est pas dans ce cas, rien ne se passe. Après cette balise de démarrage, le numéro de la broche à contrôler doit apparaître de façon à commander la broche souhaitée. Comme nous contrôlons des broches numériques normales et des broches PWM, le caractère suivant définit le mode (D, A ou B) qui détermine comment la broche doit être commandée. Enfin, le niveau demandé est ajouté à la fin du flux. Étudions l'exemple suivant.

Cette combinaison de caractères permet de programmer la broche numérique 3 à un niveau HIGH, donc la LED correspondante sera allumée. L'entrée sur le moniteur série est la suivante.

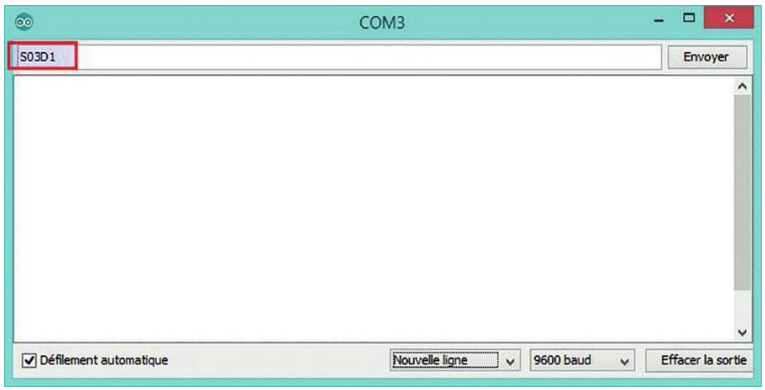


Figure 19-7
Entrée des données dans le moniteur série

Faites attention au réglage en bas de la fenêtre qui provoque un saut de ligne après l'entrée envoyée.

Voyons maintenant comment le sketch traite cette chaîne de caractères qui lui est transmise par le moniteur série. Voici le code complet de la fonction, que je vais expliquer en détail.

```
void readSerial() {
  if(Serial.available() > 0) {
    while (Serial.peek() == 'S') {        // Attente du caractère de
                                          // départ (Start)
      Serial.read();                      // S(Start)lecture du
                                          // flux dans le tampon série
      pin = Serial.parseInt();            // extraction numéro de broche
      mode = Serial.read();
// extraction mode digital D
      level = Serial.parseInt();          // extraction du niveau
      if(mode == 'D') {
        pinBlinkArray[pin - 2] = false;   // Arrêt clignotement
        digitalWrite(pin, level);         // Commande de la LED
      }
      if(mode == 'A') {
        pinBlinkArray[pin - 2] = false;   // Arrêt clignotement
        analogWrite(pin, level);          // Commande de la LED
      }
      if(mode == 'B')
        pinBlinkArray[pin - 2] = level;   // Mémorisation du clignotement
    }
```

```
    while (Serial.available() > 0) {      // Vider le tampon série
      Serial.read();
    }
  }
  // Clignotement
  if((millis() - prev) > interval) {
    prev = millis();
    ledStatus = !ledStatus;               // Inversion du statut
    for(int i = 0; i < ARRAY_SIZE; i++) {
      if(pinBlinkArray[i] == true)
        digitalWrite(pinArray[i], ledStatus);
    }
  }
}
```

Au début, la ligne

```
if(Serial.available() > 0) {...}
```

vérifie s'il y a un flux de données disponible dans le tampon de l'interface série. Si c'est le cas, le code entre les crochets sera exécuté. La méthode `peek` examine le premier caractère du flux de données sans le retirer du tampon.

```
while(Serial.peek() == 'S') {...}
```

Cette instruction garantit que le premier caractère pour contrôler les LED est bien l'étiquette de début S. Si c'est le cas, le caractère est lu dans le tampon et supprimé. Puis le ou les caractères suivants du flux de données sont analysés par la méthode `parseInt` et convertis en une valeur entière mémorisée dans la variable `pin`.

```
pin = Serial.parseInt();
```

Cela permet aux numéros de broche à un ou deux chiffres d'être correctement identifiés. Ensuite, il faut savoir comment la broche en question doit être traitée : comme une broche numérique avec un niveau LOW ou HIGH, ou comme une broche analogique à contrôler via un niveau PWM. Ceci est mémorisé dans la variable `mode`.

```
mode = Serial.read();
```

La méthode `read` lit le caractère suivant du flux de données et le place dans la variable `mode`. La lecture du flux de données se termine ensuite avec la ligne

```
level = Serial.parseInt();
```

où, à nouveau, une valeur entière valide est extraite, indiquant le niveau demandé qui est mémorisé dans la variable `level`. L'évaluation des données extraites intervient ensuite.

```
if(mode == 'D') {
  pinBlinkArray[pin - 2] = false; // Arrêt clignotement de la broche
  digitalWrite(pin, level);       // Commande de la LED
}
if(mode == 'A') {
  pinBlinkArray[pin - 2] = false; // Arrêt clignotement de la broche
  analogWrite(pin, level);        // Commande de la LED
}
if(mode == 'B')
  pinBlinkArray[pin - 2] = level; // Mémorisation du clignotement
                                  // de la LED
```

Pour les modes D et A, le tableau `pinBlinkArray` est mis à la valeur `false` pour la broche correspondante afin de stopper l'état éventuellement clignotant de cette broche, car la LED reliée à une broche ne doit clignoter que pour le mode B. Si le mode B a été identifié (clignotement), le tableau `pinBlinkArray` est mis à la valeur `level` (1 soit `true`) afin que le clignotement soit alors initialisé. Nous verrons bientôt comment cela fonctionne. Enfin, tous les caractères suivants du flux de données qui ne nous intéressent pas sont effacés par la ligne :

```
while (Serial.available() > 0) { Serial.read(); }
```

Étudions maintenant le clignotement, qui est géré par le code suivant.

```
// Clignotement
if((millis() - prev) > interval) {
  prev = millis();
  ledStatus = !ledStatus; // Inversion du statut
  for(int i = 0; i < ARRAY_SIZE; i++) {
    if(pinBlinkArray[i] == true)
      digitalWrite(pinArray[i], ledStatus);
  }
}
```

Vous avez déjà rencontré ce processus de clignotement dans le montage n° 4. Il n'utilise pas la fonction `delay`, sinon aucune autre commande ne serait acceptée durant le clignotement, puisque l'appel de la fonction `delay` provoquerait la pause du programme. Si ce processus n'est pas clair dans votre esprit, retournez au montage n° 4. À l'intérieur de la boucle `for`, le contenu du tableau `pinBlinkArray` est interrogé de façon permanente, et si une broche a la valeur `true` (vraie), elle reçoit une valeur de statut opposée via la variable `ledStatus`, grâce à la ligne suivante :

```
ledStatus = !ledStatus;
```

Mais n'avions-nous pas dit que nous devions effectuer le contrôle par un script Python ? C'est certainement plus intéressant que d'entrer les commandes dans le moniteur série, non ?

C'est vrai, mais je voulais d'abord vous montrer les bases de notre protocole. Nous sommes maintenant prêts en effet à passer à un programme externe pour envoyer les commandes. Pour programmer en Python, j'utilise l'environnement de développement gratuit PyCharm avec la version 3.10 de Python et le module pyserial. Voici les liens utiles pour télécharger ces logiciels :

- PyCharm : https://www.jetbrains.com/pycharm/download/
- pyserial : https://pypi.python.org/pypi/pyserial

Cet ouvrage n'étant pas un livre sur la programmation avec Python, je vais seulement détailler les commandes utilisées ici. Pour plus d'information, référez-vous aux manuels sur Python ou consultez Internet.

Figure 19-8 ▶
Script Python permettant le contrôle des LED

```
1   import serial
2
3   port = serial.Serial(port='COM17', baudrate=38400)
4   print(port.isOpen())
5   cmd = ['S02D1\r', 'S03A80\r',
6          'S04B1\r', 'S05D1\r',
7          'S06D1\r', 'S07D1\r',
8          'S08B1\r', 'S09A10\r',
9          'S10D1\r', 'S11D1\r',
10         'S12D1\r', 'S13D1\r']
11
12  while True:
13      if port.isOpen():
14          port.writelines(cmd)
```

À la ligne 1, le module pyserial requis est importé, car nous voulons accéder à l'interface série. La ligne 3 instancie un objet série nommé `port` qui contient le paramètre de l'interface COM et le débit de transmission. Le taux de transfert doit correspondre à celui que vous avez indiqué dans le sketch Arduino, sinon il n'y aura pas de communication entre Python et votre Arduino. À la ligne 4, à l'aide de la méthode `isOpen`, j'affiche sur la console l'état du port de l'interface mentionnée afin de vérifier qu'il a bien été ouvert. La variable `cmd` utilisée en Python contient une liste des informations de commande pour les broches individuelles. Je laisse certaines LED s'allumer complètement mais deux LED sont contrôlées par un niveau de modulation PWM et deux autres clignotent. La boucle `while` de la ligne 12 exécute sans fin le code qui suit et utilise la méthode `writelines` pour

envoyer toutes les définitions de broche contenues dans la liste à l'interface série. Si vous souhaitez piloter une seule broche, vous pouvez limiter la liste à un seul élément ou utiliser directement la méthode `write`.

```
while True:
    if port.isOpen():
        port.write('S02D1')
```

◀ Figure 19-9
Script Python pour piloter une seule LED

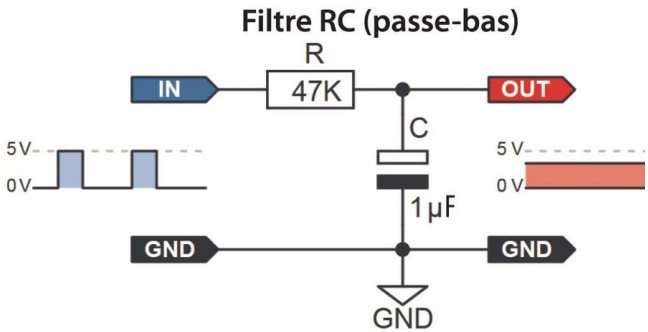
Problèmes courants

Si le contrôle de vos sorties numériques ne fonctionne pas, vérifiez les points suivants.

- Assurez-vous que le câblage ait été bien réalisé.
- Vérifiez qu'il n'y a aucun court-circuit dans le câblage.
- Veillez à ce que l'émetteur et le récepteur aient la même vitesse de transmission.
- Faites attention à l'emploi différencié des majuscules et des minuscules.

Utilisation d'un filtre passe-bas

Vous avez beaucoup appris sur le signal PWM et ses effets sur un composant connecté. La luminosité d'une LED peut être réglée et la vitesse d'un moteur contrôlée. Mais savez-vous que vous pouvez aussi générer une tension continue à partir de ce On/Off qui varie en fonction du cycle du signal ? Comment est-ce possible ? Vous avez juste besoin d'une résistance et d'un condensateur branchés comme suit.



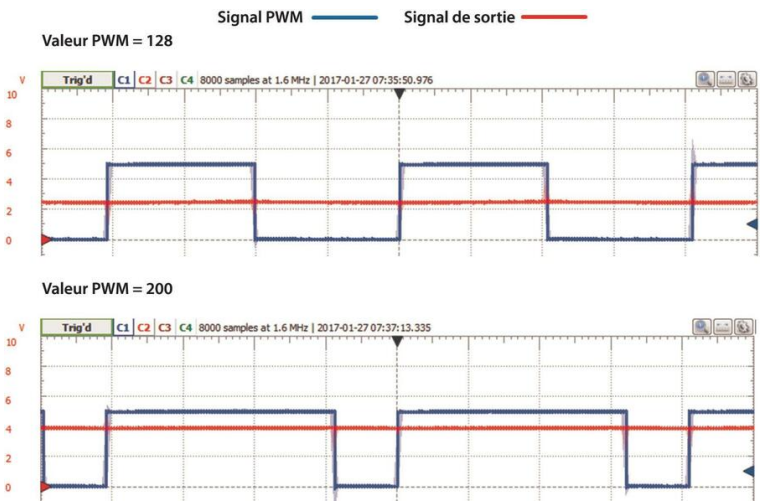
◀ Figure 19-10
Le circuit filtre RC

Si vous regardez ce circuit composé d'une résistance et d'un condensateur, cela doit vous faire penser à un diviseur de tension que nous avons

déjà étudié. En effet, ce circuit RC n'est ni plus ni moins qu'un diviseur de tension dépendant de la fréquence, qui dans cette configuration constitue ce que l'on appelle un *filtre passe-bas*. Il se nomme ainsi car il laisse passer les basses fréquences plutôt que les hautes. Ce circuit est encore appelé *intégrateur*, assurant le lissage du signal d'entrée. Le condensateur est placé en parallèle à la sortie et permet un stockage d'énergie. Si la tension à l'entrée est de 5 V par exemple, le condensateur se charge à travers la résistance et le signal de sortie augmente lentement. Lorsque le niveau d'entrée revient à 0 V, le condensateur se décharge à travers la résistance, si bien que le signal de sortie ne retombe pas brusquement à 0 V. Si ce changement se produit assez rapidement, le condensateur va donc lisser le signal d'entrée. Bien entendu, ce comportement dépend des caractéristiques des composants R et C.

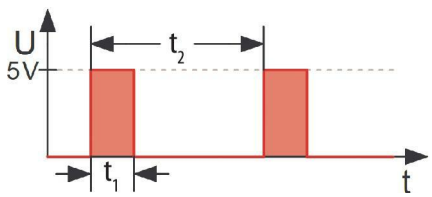
Étudions deux signaux PWM différents et le lissage correspondant.

Figure 19-11 ▶
Filtrage RC passe-bas



Vous pouvez constater qu'avec un niveau de signal PWM plus élevé, la tension de sortie est plus haute.

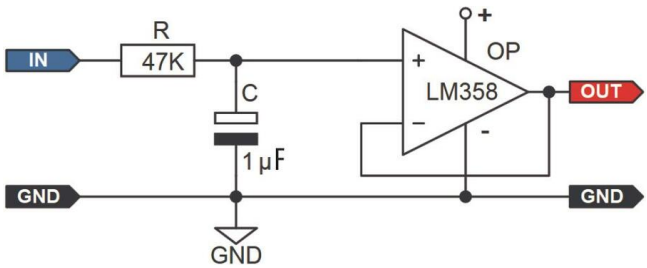
Mais, peut-on réellement calculer la tension de sortie lorsque les paramètres sont connus ? Eh bien, oui, c'est possible ! Examinons ce graphique représentant les impulsions :



En utilisant la formule suivante, vous pouvez calculer la tension de sortie U_S :

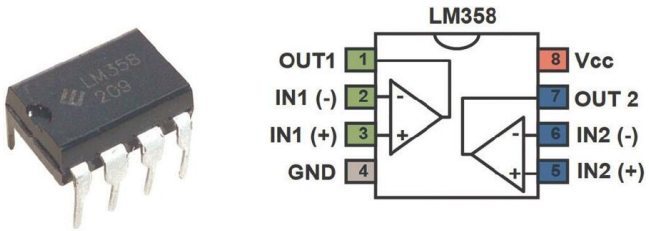
$$U_s = \frac{t_1 \cdot 5\,\mathrm{V}}{t_1 + t_2}$$

Si vous avez un oscilloscope, utilisez cette formule pour faire quelques calculs et vous verrez que le résultat est correct. Naturellement, dans le cas d'un seul circuit RC, sa capacité de stockage sera très faible : aussi, dans une application pratique, un petit amplificateur devra être utilisé sous la forme d'un amplificateur opérationnel. Le circuit suivant montre cet amplificateur opérationnel (par exemple, le LM358).



◀ **Figure 19-12**
Le filtre RC avec amplificateur opérationnel LM358

Le circuit intégré LM358 se présente sous forme d'un petit boîtier comportant 8 broches dont le brochage est le suivant.



◀ **Figure 19-13**
Affectation des broches de l'amplificateur opérationnel LM358

Pour terminer, voici une formule permettant de calculer la fréquence de coupure d'un filtre passe-bas :

$$f_C = \frac{1}{2 \cdot \pi \cdot R \cdot C}$$

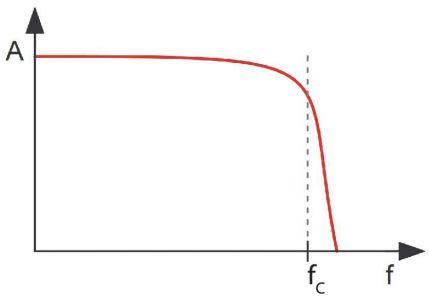
Vous pourrez trouver des informations complémentaires sur ce sujet en consultant ce lien :

https://fr.wikipedia.org/wiki/Filtre_passe-bas

Mais pourquoi le circuit RC ne laisse passer que les basses fréquences ? Je vous explique. Nous savons que la résistance d'un condensateur est très

élevée pour un courant continu. Le condensateur laisse passer le courant uniquement pendant la phase de charge puis le bloque lorsqu'il est chargé. Si le condensateur est inversé ou déchargé, un courant circule à nouveau jusqu'à la décharge complète. Cependant, plus ces deux processus de charge et de décharge ont lieu rapidement, plus la réactance du condensateur est faible et ne constitue pas un obstacle au courant. À proprement parler, il n'y a pas de courant car l'énergie n'est absorbée que pendant le processus de charge et libérée à nouveau pendant le processus de décharge. Puisque dans le circuit RC, le condensateur est placé en parallèle à la sortie et qu'il possède une résistance plus ou moins faible selon que la fréquence du signal est plus ou moins élevée, les fréquences élevées sont plus ou moins bloquées. Pour mieux comprendre ce comportement, examinez le graphique suivant où la fréquence est en abscisse et la réponse en amplitude en ordonnée.

Figure 19-14 ▶
Courbe caractéristique d'un circuit passe-bas



On voit également où se situe la fréquence de coupure f_c.

Qu'avez-vous appris?

- Vous avez vu comment utiliser un protocole de transfert pour envoyer différentes données sur le port série afin de contrôler votre carte Arduino.
- Cela peut être fait via une saisie dans un programme terminal tel que le moniteur série, un script Python ou une application ergonomique dotée d'une interface graphique. Le programme C# que nous avons utilisé n'est qu'un exemple d'implémentation possible car vous pouvez utiliser n'importe quel autre langage de programmation qui fournit l'accès au port série.
- Vous avez étudié un circuit composé d'une résistance et d'un condensateur, qui fonctionne comme un filtre passe-bas et convertit un signal PWM en un signal analogique. Ce circuit est plus robuste s'il est complété, en aval, par un amplificateur opérationnel.

Communication sans fil par Bluetooth

Cela fait déjà quelques années que sonne le glas du câble de transmission. Depuis qu'une fonction Bluetooth a été intégrée à la grande majorité des smartphones récents, ce protocole de transmission de données sur de courtes distances s'est imposé et il est devenu incontournable. Cela concerne évidemment aussi le domaine du bidouillage électronique où le Bluetooth a déjà sa place depuis longtemps. Il est donc grand temps de l'intégrer à un montage.

Qu'est-ce que la communication radio ?

Jusqu'ici, l'échange de données, c'est-à-dire la communication entre l'ordinateur et la carte Arduino, a toujours été effectué de manière filaire par l'intermédiaire du port série. C'est amplement suffisant pour la majorité de nos expériences, car rien ne s'y oppose. Mais imaginez que vous vouliez que votre robot s'éloigne un peu et que la longueur du câble USB soit insuffisante. Comment feriez-vous ? De même, s'il devait effectuer de nombreuses rotations dans différentes directions, le câble finirait par être très emmêlé. Une solution consisterait à utiliser un câble USB plus long, mais la longueur maximale est généralement de cinq mètres. Tout ce qui dépasse présente un risque et les erreurs sont alors très difficiles à localiser. Il me paraît être grand temps d'envisager la communication radio. Cette forme de communication sans fil facilite considérablement les échanges de données avec le destinataire final sans qu'il soit nécessaire de poser des câbles dans lesquels on finit toujours par s'empêtrer. Certes, il y a toujours des restrictions concernant la portée qui varie en fonction de la puissance de l'émetteur. Le WLAN, qui est probablement déjà proposé par votre routeur, est l'une des liaisons radio les plus connues. Ce réseau a une portée de quelques mètres qui varie en fonction de la nature de l'environnement. La portée de 300 mètres évoquée par certains me paraît uniquement

possible en l'absence d'obstacles et dans des conditions très favorables, qui sont difficiles à remplir à l'intérieur de bâtiments. Dans tous les cas, la communication passera mieux entre deux pièces que d'un étage à l'autre, surtout lorsque le plancher contient une grille de blindage sous forme d'armatures en acier. La portée se situe alors entre 10 à 20 mètres.

Nous allons étudier un procédé de transmission radio qui est utilisé sur de courtes distances ne dépassant pas 100 mètres : le Bluetooth. Comme le Bluetooth et WLAN se partagent la même bande de fréquence de 2,4 GHz, cela peut engendrer des perturbations dues à des recouvrements. Ne soyez donc pas étonné si tout ne fonctionne pas du premier coup. En cas de problème, éloignez-vous de quelques mètres ou réessayez après avoir désactivé le WLAN. Quelques essais sont parfois nécessaires, mais vous finirez toujours par réussir à établir une connexion. Montrez-vous persévérant !

La portée du Bluetooth dépend aussi de la puissance de l'émetteur qui peut appartenir aux trois classes suivantes :

Tableau 20-1 ▶ Classes Bluetooth avec puissance et portée

Classe	Puissance	Portée
1	100 mW	env. 100 mètres
2	2,5 mW	env. 10 mètres
3	1 mW	env. 1 mètre

L'émetteur n'est évidemment pas le seul responsable de la portée limitée. Le récepteur y contribue aussi. Des facteurs tels que la qualité de l'antenne et la sensibilité du récepteur jouent aussi un rôle important. Si vous souhaitez obtenir plus d'informations à ce sujet, je vous invite à faire des recherches sur Internet ou à lire la littérature spécialisée. J'aimerais vous présenter un shield intéressant que nous utiliserons pour la communication sans fil avec la carte Arduino. Voyons de quels composants vous allez avoir besoin pour ce montage.

Composants nécessaires

Ce montage nécessite les composants suivants.

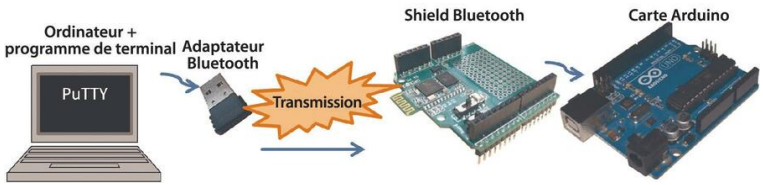
Tableau 20-2 ▶ Liste des composants

Composant	
1 shield Bluetooth de ITead Studio, version 2.1	
1 adaptateur Bluetooth (si votre ordinateur portable n'en est pas équipé)	

Mais comment la communication est établie et à l'aide de quels composants ? Comment interroger les données reçues sur l'Arduino ? Existe-t-il aussi une bibliothèque spéciale ? Ces questions sont très pertinentes et je me les suis également posées. C'est assez simple. Il est possible de communiquer avec n'importe quel appareil Bluetooth, que ce soit un adaptateur Bluetooth, par exemple, qui est branché sur l'ordinateur ou un smartphone.

◀ Figure 20-1
Adaptateur Bluetooth pour port USB

Ce qu'il y a de vraiment génial dans la communication avec une carte Arduino, c'est le fait qu'elle s'effectue entièrement par le biais du port série de la carte et qu'aucune bibliothèque particulière n'est donc requise. De plus, après avoir installé l'adaptateur Bluetooth sur votre ordinateur, vous disposez d'un nouveau port COM. Peut-être voyez-vous déjà où je veux en venir ? Lorsque vous connectez un programme de terminal, comme *PuTTY* au port COM de l'adaptateur Bluetooth et que vous y tapez des commandes, celles-ci sont envoyées sur les ondes par le biais du Bluetooth. Par conséquent, si votre carte Arduino est reliée au module Bluetooth, les commandes envoyées par PuTTY peuvent y être reçues et analysées par le biais du port série. Sur la figure suivante, j'ai tenté de représenter le cheminement des données, de gauche à droite, tel qu'il s'effectue durant la communication.

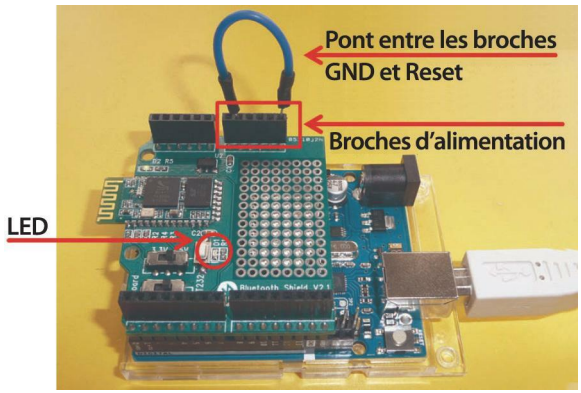


◀ Figure 20-2
L'adaptateur Bluetooth émet et le shield Bluetooth reçoit.

Utilisation d'un adaptateur Bluetooth

Si votre ordinateur n'est pas équipé d'un adaptateur Bluetooth – les ordinateurs portables récents en intègrent généralement un, alors vous devez vous en procurer un. Si vous utilisez Windows 7 ou une version ultérieure, le *stack* (ou pile) Bluetooth, c'est-à-dire le programme de communication avec ce support, est préinstallé. Vous n'avez donc rien de plus à faire. L'installation s'effectue en arrière-plan lorsque vous raccordez le module sur un port USB disponible. Toutefois, vous devez ajouter un nouvel appareil Bluetooth dans votre ordinateur, car le shield Bluetooth doit être identifié par le système d'exploitation. Nous le ferons ensemble. Mais avant, nous allons établir une connexion entre le moniteur série et le shield Bluetooth afin de procéder à quelques essais et ajustements. Pour commencer, branchez le shield sur la carte Arduino.

Figure 20-3 ▶
Shield Bluetooth enfiché

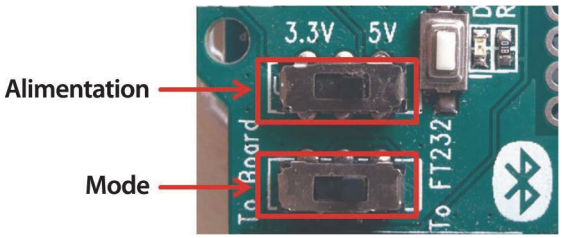


Raccordez l'entrée Reset, qui est active au niveau LOW, à la masse de la carte.

ATTENTION AU RACCORDEMENT !

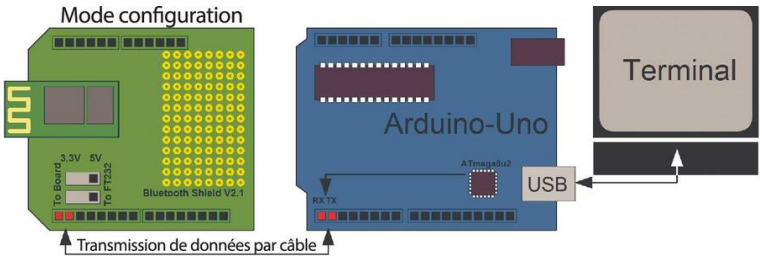
Comme vous créez un pont, faites très attention à bien raccorder la broche Reset à la broche GND ! À la moindre erreur, vous risquez de produire un court-circuit.

Vous devez préalablement avoir connecté le shield Bluetooth. Mais avant cela, examinez attentivement la rangée de broches correspondantes. Lorsque vous avez créé le pont comme illustré, la LED *D1* de la carte se met à clignoter rapidement. C'est bon signe ! Le shield Bluetooth dispose de deux petits interrupteurs coulissants.



◀ Figure 20-4
Configuration du shield Bluetooth

Comme vous utilisez l'Arduino Uno, l'interrupteur du haut doit être en position 5 V. Lors de la configuration de la carte, l'interrupteur du bas doit être en mode To FT232, ce qui signifie que la transmission des données s'effectue au moyen de l'ATmega8U2.

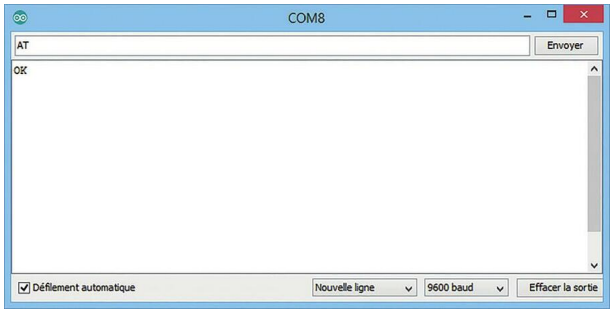


◀ Figure 20-5
Mode configuration du shield Bluetooth

Dans ce cas, la carte Arduino est utilisée comme interface entre un programme de terminal, comme le moniteur série ou PuTTY, et le shield Bluetooth. Vérifiez que vous avez bien sélectionné le port COM de la carte Arduino, car la communication avec le shield s'effectue encore de façon filaire. Une autre intervention est requise de votre part, car le port série ne doit pas être bloqué par l'Arduino. Raccordez la broche Reset à la masse afin que le microcontrôleur se trouve en mode de réinitialisation. Ensuite, la configuration du shield peut commencer. Ouvrez le moniteur série. La communication avec le shield Bluetooth s'effectue au moyen de commandes *AT* (*AT* signifie *Attention*) qui, autrefois, servaient à configurer les modems. Ce protocole a perduré jusqu'à aujourd'hui et il est encore utilisé pour la configuration de divers appareils. Pour vous assurer que le shield Bluetooth est correctement raccordé, saisissez `AT`, puis appuyez sur la touche Entrée. Vérifiez au préalable que la vitesse de transmission a bien été réglée sur 9 600 bauds et que l'option *Pas de fin de ligne* a été sélectionnée.

Réponse du shield Bluetooth :

Figure 20-6 ▶
Réponse du shield BT

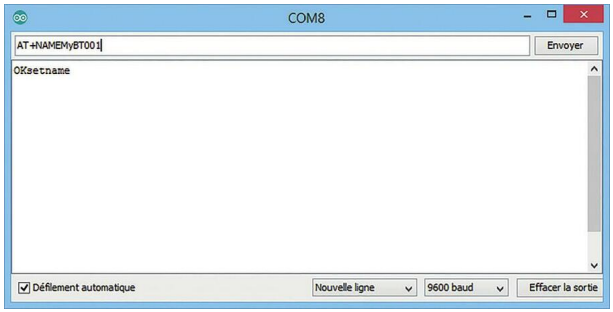


Si tout fonctionne correctement, le shield Bluetooth répond `OK`.

Nommer le shield BT

Vous devez maintenant nommer le module au moyen de la commande `AT+NAME` suivie du nouveau nom.

Figure 20-7 ▶
Nouveau nom du shield BT



Je l'ai nommé `MyBT001`. Vérifiez qu'il n'y a pas d'espace entre la commande AT et l'action que vous voulez exécuter. Le module confirme l'attribution du nom par `OKsetname`. Le code Pin qui protège par défaut la communication contre les accès indésirables est 1234. Personnalisez-le également.

Modifier le code Pin du shield BT

Saisissez la commande `AT+PIN` puis saisissez le nouveau code Pin afin de définir un nouveau *code de jumelage*.

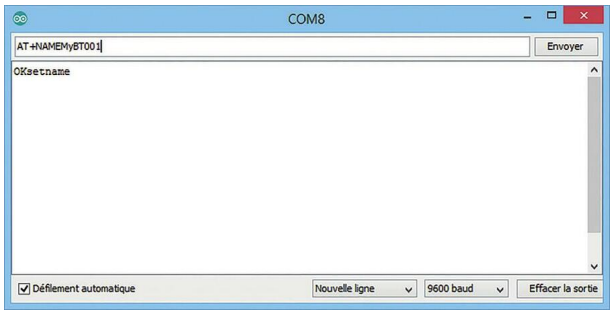


Figure 20-8
Nouveau code de jumelage du shield BT

Modifier la vitesse de transmission du shield BT

Vous pouvez évidemment aussi modifier la vitesse de transmission qui est définie par défaut sur 9 600 bauds. Utilisez à cette fin la commande **AT+BAUDx** suivie de la nouvelle vitesse de transmission. Le petit `x` est un code qui correspond à une certaine vitesse qui est indiquée dans le tableau suivant :

Tableau 20-3
Vitesses de transmission

Paramètre	Vitesse de transmission
BAUD1	1 200
BAUD2	2 400
BAUD3	4 800
BAUD4	9 600
BAUD5	19 200
BAUD6	38 400
BAUD7	57 600
BAUD8	115 200

Même si je conserve le débit prédéfini de 9 600 bauds, je saisis la commande suivante pour en illustrer le fonctionnement :

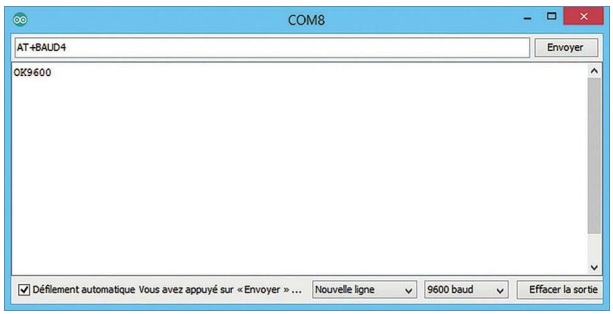


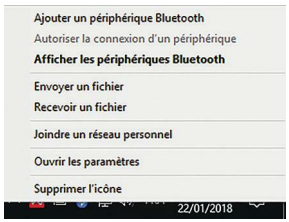
Figure 20-9
Nouvelle vitesse de transmission du shield BT

Le message `OK9600` indique que le débit a bien été défini sur 9 600 bauds.

Ajout d'un nouveau shield BT

Maintenant que le shield BT a été configuré, il va pouvoir être identifié par le système d'exploitation. Cliquez sur l'icône bleue Bluetooth avec le bouton gauche de la souris pour afficher le menu suivant.

Figure 20-10 ▶
Ajouter un nouveau périphérique Bluetooth



Sélectionnez la commande *Ajouter un périphérique*. La boîte de dialogue qui apparaît énumère tous les adaptateurs Bluetooth qui sont installés dans le système d'exploitation. Le shield BT y figure sous son nouveau nom. Cliquez sur *Suivant*. Ensuite, vous devez sélectionner un mode de jumelage. Si vous utilisez un code, choisissez l'option avec code et saisissez le code de jumelage dans le champ correspondant. Un message devra apparaître, vous confirmant que l'installation a réussi.

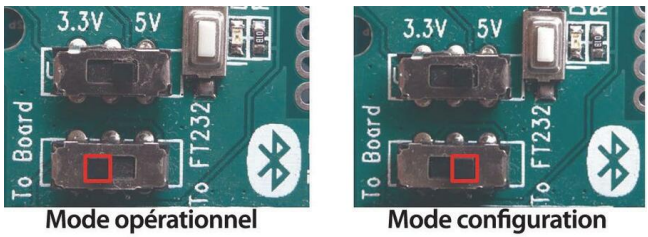
L'appareil apparaît aussi dans le Gestionnaire de périphériques Windows. Vérifiez qu'un port COM lui a bien été attribué, car c'est par ce biais que s'effectuera la communication avec le shield BT. Notez que les ports COM de votre ordinateur ne seront pas forcément les mêmes.

Mais n'est-il pas possible d'utiliser l'Arduino-Talker présenté dans le montage précédent pour la communication Bluetooth ? Cela nous éviterait d'avoir à saisir ces lignes de commandes compliquées. Bien sûr, c'est parfaitement possible !

Réalisation du circuit

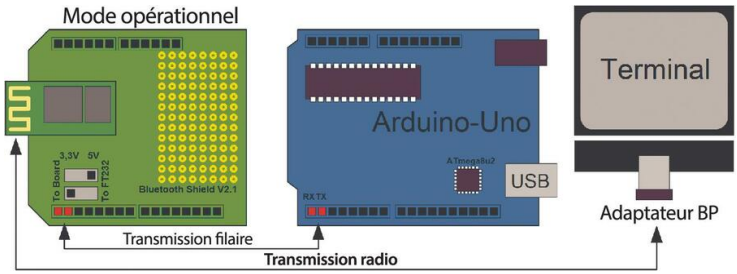
Nous allons maintenant pouvoir tester le shield Bluetooth. Vous aurez besoin du code du sketch du montage précédent sur l'Arduino-Talker. Au préalable, vous devez évidemment supprimer la réinitialisation au moyen du petit câble de patch et faire passer la carte du mode configuration au

mode opérationnel. Faites glisser le curseur vers la gauche en position *To Board* :



◀ **Figure 20-11**
Les deux modes du shield BT

Les données circulent alors comme illustré sur la figure suivante :



◀ **Figure 20-12**
Mode opérationnel du shield Bluetooth

Le transfert des données s'effectue maintenant par liaison radio. C'est ainsi qu'elles sont acheminées jusqu'au port série de la carte Arduino en passant par le *shield Bluetooth*.

À propos du schéma du mode opérationnel, le programme de terminal doit bien être connecté à un port COM ? Est-ce le même que celui utilisé par le moniteur série pour la communication avec l'Arduino ?

Bien sûr que non ! Le port COM du moniteur série est directement raccordé à la carte Arduino. Il s'agit d'une communication filaire. Ici, nous voulons communiquer au moyen d'une liaison radio et c'est pour cette raison que nous avons besoin d'un adaptateur Bluetooth. Cet adaptateur détecte les périphériques Bluetooth – comme le shield Bluetooth – qui se trouvent à proximité et les ajoute à votre système afin que vous puissiez y accéder par une liaison sans fil. Si l'ajout du shield Bluetooth a réussi, un nouveau port COM apparaît sur votre ordinateur. Vous pouvez alors établir le contact avec votre programme de terminal par le biais de ce port qui représente

le shield Bluetooth. Nous en sommes là. Vous pouvez continuer à utiliser votre moniteur série pour solliciter le shield Bluetooth par liaison radio. Il vous suffit de sélectionner le port COM correspondant à l'aide de la commande *Outils* | *Port*.

Nous voulons maintenant procéder au test de l'ensemble. Chargez le sketch de l'Arduino-Talker sur la carte Arduino. Pensez à vérifier que la vitesse de transmission du sketch et celle du shield BT concordent. Elles doivent toutes les deux être réglées sur 9 600 ou 38 400 bauds, par exemple. Essayez différentes vitesses afin de déterminer lesquelles sont possibles. Si vous rencontrez des problèmes lors du téléversement, déconnectez provisoirement le shield BT de la carte Arduino. Une fois que tout a fonctionné correctement, enfichez la shield des LED du montage de l'Arduino-Talker – si vous l'avez fabriqué – sur le shield BT.

Voici ce que cela donne :

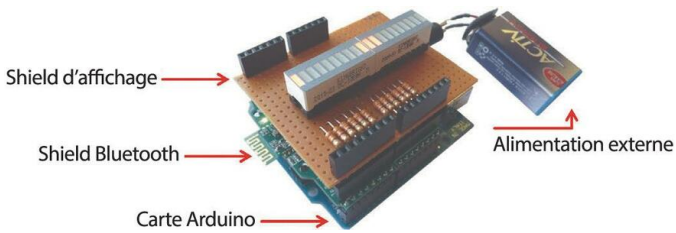


Figure 20-13 ▶ Shield BT et shield des LED enfichées sur la carte Arduino

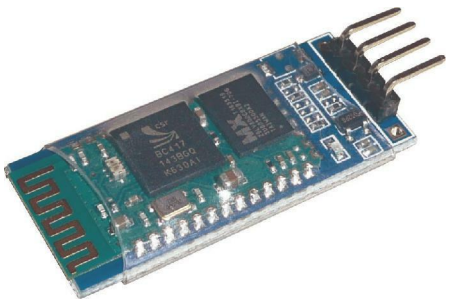
Comme vous pouvez le constater, après le téléversement du sketch Arduino-Talker, la liaison USB est superflue et vous pouvez la supprimer. Pour finir, vous pouvez connecter une batterie de 9 V à la prise de l'alimentation en tension externe. La carte Arduino devrait être autonome et ne plus recevoir que les instructions par Bluetooth. Jetez un dernier coup d'œil au Gestionnaire de périphériques. Sélectionnez le bon port COM dans l'IDE Arduino et ouvrez le moniteur série afin de choisir la vitesse de transmission correcte et la configuration *Saut de ligne (CR)*. Ensuite, vous pouvez saisir une commande. Les LED correspondantes doivent clignoter sur le shield des LED.

Au lieu du port COM Arduino, connectez-vous au port COM Bluetooth, comme je viens de le faire avec le moniteur série. Fermez préalablement le moniteur série, sinon le port COM sera occupé par ce programme. La communication avec l'Arduino-Talker s'effectue alors par liaison radio.

▲ **Figure 20-14**
Commande du shield Bluetooth à l'aide de l'application Arduino-Talker

Le module Bluetooth HC-06

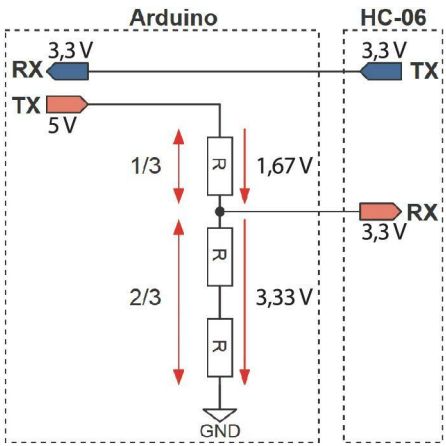
Il existe d'autres modules Bluetooth intéressants et bon marché. Parmi eux, j'aimerais vous présenter le module HC-06.



◀ **Figure 20-15**
Le module Bluetooth HC-06

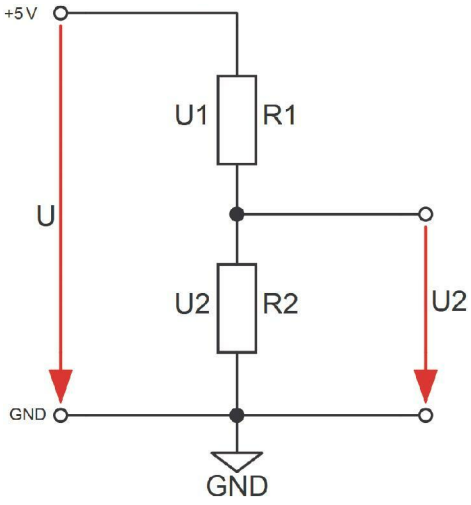
Il permet également d'établir des liaisons Bluetooth. Toutefois, bien que ce module possède une alimentation en tension de +5 V, les lignes de données RX/TX du port série fonctionnent en 3,3 V. Même si cela ne devrait pas poser de problèmes, c'est néanmoins risqué. Voici un petit circuit comportant deux résistances qui font passer le niveau de +5 V à +3,3 V.

Figure 20-16 ▶
Passage du niveau de +5 V à +3,3 V



La ligne émettrice TX de l'Arduino fonctionne en +5 V, ce qui est trop élevé pour le module HC-06. La ligne émettrice TX de ce dernier fournit seulement 3 V, ce qui suffit à l'Arduino et ne nécessite donc pas d'ajustements. Le diviseur de tension illustré permet de ramener la tension de +5 V à 3,3 V. Revoyons le principe du diviseur de tension.

Figure 20-17 ▶
Un diviseur de tension



La tension d'entrée *U* de +5 V se trouve sur le côté gauche et la tension de sortie U_2 se trouve du côté droit. Cette dernière n'a pas encore véritablement de valeur, puisque les deux résistances R_1 et R_2 n'ont pas encore non plus de valeurs définies. La tension de sortie U_2 est appliquée à la résistance R_2, ce qui signifie qu'une tension est ajustée entre les deux résistances. Une partie de la tension d'alimentation *U*, qui est de +5 V, chute à travers de R_1, et l'autre à travers R_2. Pour que la tension requise de 3,3 V chute à travers

de R_2, elle doit chuter de 1,7 V à travers R_1. Les rapports de tension sont équivalents à ceux de la résistance et nous pouvons donc établir la formule suivante : la tension totale (U) rapportée à la résistance totale ($R_1 + R_2$) est égale à la tension partielle (U_2) rapportée à la résistance partielle (R_2). Traduit en langage mathématique, cela donne :

$$\frac{U}{R_1 + R_2} = \frac{U_2}{R_2}$$

Il ne reste plus qu'à déplacer R_2 et on obtient le résultat suivant :

$$U_2 = \frac{R_2}{R_1 + R_2} \cdot U$$

Si vous choisissez les valeurs de résistance suivantes, par exemple, vous obtenez un résultat plausible :

$$R_1 = 1{,}2\text{K et } R_2 = 2{,}2\text{K}$$

$$U_2 = \frac{2{,}2\text{K}}{1{,}2\text{K} + 2{,}2\text{K}} \cdot 5\text{ V} = 3{,}24\text{ V}$$

La valeur de 3,24 V est très proche des +3,3 V requis et elle permet donc de travailler. Même si vous ne disposez pas des deux valeurs de résistance citées, ce n'est pas dramatique, car trois valeurs identiques sont nécessaires. Le diviseur de tension peut aussi être fabriqué avec des résistances de 1K.

Exercice complémentaire

Essayez de diffuser différents sons par une liaison Bluetooth avec un élément piézoélectrique. Peut-être devrez-vous légèrement adapter le protocole. Écrivez le vôtre, car vos idées différeront sans doute des miennes. L'essentiel est que le circuit et le sketch forment un tout et qu'ils soient sur la même longueur d'onde.

Problèmes courants

Vérifiez ce qui suit en cas de problème de commande des sorties numériques.

- Le câblage est-il correct ?
- Pas de court-circuit éventuel ?
- Vérifiez que vous utilisez le port COM adéquat. Cela m'est déjà arrivé et j'ai mis du temps à localiser l'origine du problème.

- Si vous utilisez un autre programme de terminal à la place du moniteur série, comme PuTTY, pour la saisie, assurez-vous d'abord qu'il est correctement configuré. Il est parfois nécessaire de faire des essais afin de trouver les bons paramètres.

Qu'avez-vous appris ?

- Vous avez appris à établir une connexion sans fil Bluetooth.
- Nous avons réutilisé les commandes générées par le montage précédent dans le moniteur série comme entrée du shield Bluetooth pour commander l'affichage.

Montage 21

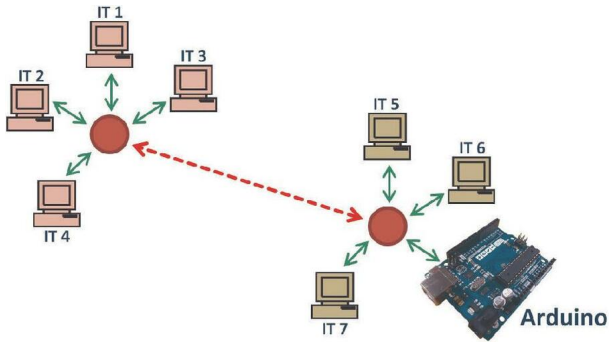
Communication réseau

Avant de connecter votre carte Arduino Uno à votre réseau domestique ou à Internet, vous devez savoir quelques petites choses. Dans ce montage, nous allons donc étudier les bases de la communication réseau. À l'aide d'un shield complémentaire connecté à votre Arduino Uno, vous raccorderez la carte à Internet au moyen d'un câble Ethernet. Une fois la connexion établie, le microcontrôleur transmettra des données qui seront visualisées sur une page web. Nous profiterons également de ce chapitre pour dresser une rapide introduction au langage HTML.

Qu'est-ce qu'un réseau ?

Le plus gros réseau que l'Homme utilise quotidiennement est le *World Wide Web*. Il s'agit de l'interconnexion d'une multitude de systèmes informatiques en contact les uns avec les autres dans le monde entier. On parle de réseau dès l'instant où deux ordinateurs sont associés via un support de transmission approprié (par exemple : câble Ethernet, fibre optique ou WLAN). Vous pouvez l'imaginer comme un cerveau contenant plusieurs centaines de milliards de cellules nerveuses. Chacune d'elles possède jusqu'à dix mille synapses. Ce sont des voies de communication qu'elles utilisent pour transmettre ou échanger des informations. Chaque cellule nerveuse du cerveau figure un ordinateur en contact avec d'autres systèmes au moyen des synapses – donc de sa carte réseau (ou de ses cartes le cas échéant).

Figure 21-1 ▶
Petit réseau avec carte Arduino

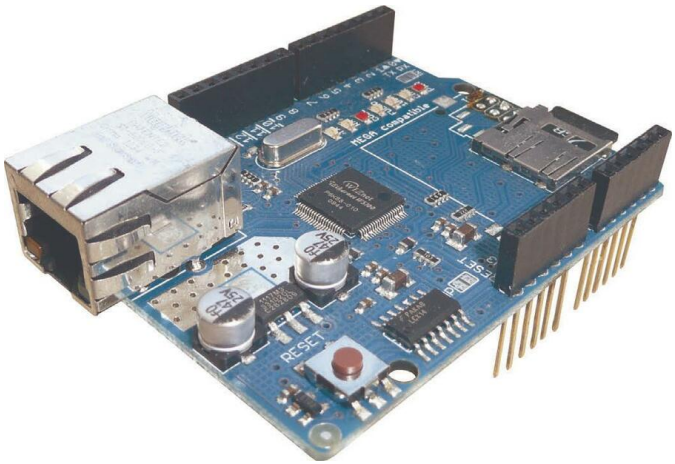


Les différents systèmes informatiques, que j'ai appelés IT1 à IT7 sur la figure 21-1 pour plus de commodité, sont reliés entre eux au moyen des cartes ou plutôt des câbles de réseau. Cette représentation est bien sûr simplifiée, car les composants du réseau sont par exemple reliés par des *switches* dans la réalité. Ces répartiteurs ou coupleurs de réseau transmettent les données de manière intelligente aux différents utilisateurs. La figure 21-2 montre un connecteur de type RJ45 d'un câble réseau couramment utilisé.

Figure 21-2 ▶
Connecteur RJ45 d'un câble de réseau

Je pense que vous avez déjà vu une fiche de cette sorte puisque votre ordinateur est relié à coup sûr par un câble de réseau au routeur qui établit une liaison vers votre fournisseur d'accès, autrement dit vers Internet.

Il n'y a pas de prise femelle pour cette fiche sur la carte Arduino car cette dernière ne dispose pas d'une connexion réseau. Un composant réseau supplémentaire est donc nécessaire.



◀ Figure 21-3
Shield Ethernet

La figure 21-3 montre un shield Ethernet, qui dispose en plus d'un socle microSD. Vous pouvez y stocker temporairement des données, mais là n'est pas le sujet. Vous trouverez de plus amples informations sur le shield Ethernet à l'adresse suivante :

https://www.arduino.cc/en/Guide/ArduinoEthernetShield

Nous avons déjà utilisé plusieurs fois le mot Ethernet. Qu'est-ce que ça veut dire ? Le moment est venu d'aborder certains points spécifiques aux réseaux pour le savoir.

Ethernet

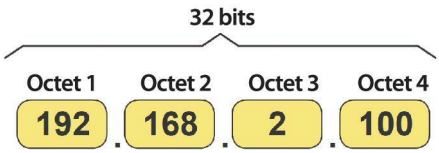
Le mot Ethernet qualifie une technologie câblée pour transmettre des données. Depuis les années 1990, elle est la norme pour toute une gamme de technologies LAN *(Local Area Network)*. Le transfert des données est, en principe, assuré par un câble à paires torsadées *(Twisted-Pair-Cable)* selon la norme CAT-5 ou supérieure.

TCP/IP

Ethernet utilise un protocole appelé TCP (*Transfer Control Protocol*, protocole de contrôle de transfert) pour transmettre des données. Ce protocole permet de transférer des informations au moyen d'un réseau local ou global et garantit une communication sans erreurs. Des mécanismes permettent, en cas d'erreur de données menaçante, de corriger ou de retransmettre les paquets de données à transférer. La désignation IP *(Internet Protocol)* concerne l'adressage des paquets de données à transférer qui doivent être acheminés de l'émetteur à un destinataire bien défini. Ce protocole assure donc l'adressage des paquets de données à transmettre. Chaque utilisateur du réseau possède une adresse précise, comparable au numéro de maison dans une rue. Pour qu'un colis puisse être par exemple livré à coup sûr par la Poste, les numéros des maisons ne doivent pas être en double, ce qui est le cas normalement. L'IP est toujours indiqué ou utilisé en rapport avec le TCP.

Adresse IP

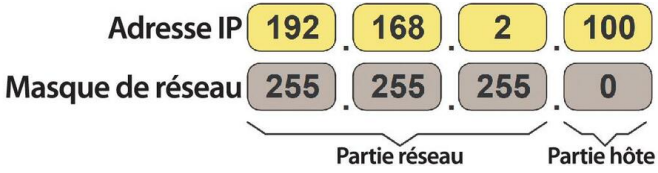
L'adresse IP d'un utilisateur doit satisfaire à l'exigence d'unicité dans un réseau. Elle est affectée à un appareil qui fait partie du réseau, garantissant ainsi qu'il est adressable et accessible. Les adresses IP de la notation Ipv4 sont composées de quatre octets (32 bits).



Cette adresse a été attribuée par mon routeur à mon PC, afin que je sois disponible sur le réseau.

Masque de réseau

Une adresse IP comprend toujours une partie réseau et une partie hôte. Le masque de réseau définit quant à lui combien d'appareils doivent être atteints dans un réseau et lesquels se trouvent dans d'autres réseaux.



Pour parvenir à la partie hôte, l'adresse IP est combinée au masque de réseau par une opération `ET`. D'après le masque précédent, il est théoriquement possible d'avoir $2^8 = 256$ ordinateurs dans le réseau. Je dis bien théoriquement, car 255, par exemple, a une signification particulière. Des détails supplémentaires sortiraient du cadre de ce livre, c'est pourquoi je vous invite à consulter la littérature spécialisée ou à regarder sur Internet.

Adresse MAC

L'adresse MAC *(Media Access Control)* doit être sans ambiguïté à l'échelle mondiale. Elle a été attribuée à chaque adaptateur de réseau. Elle se compose de six octets, les trois premiers contenant un code fabricant OUI *(Organizational Unit Identifier)*. Les trois autres octets contiennent l'indicatif de l'appareil, assigné par le fabricant en question. Voici un exemple d'adresse MAC pour une carte réseau :

```
1C-6F-65-94-D5-1A
```

Passerelle

Une passerelle (*gateway*, en anglais) est un passage vers une zone particulière qui, rapporté à notre thématique, peut être appelé *passerelle de réseau*. De quel appareil pourrait-il s'agir ? Le routeur, qui se trouve à moitié sur Internet, passe pour être une passerelle. Mon routeur a par exemple `192.168.2.1` comme adresse IP et transmet mes demandes à mon fournisseur d'accès, c'est-à-dire sur Internet. Si vous ouvrez une console DOS et que vous écrivez la commande `ipconfig /all`, vous obtenez, entre autres, les indications suivantes :

```
Passerelle par défaut . . . . . : 192.168.178.1
Serveur DHCP. . . . . . . . . . . : 192.168.178.1
```

La figure 21-4 montre le shield Ethernet combiné à vote carte Arduino.

◀ Figure 21-4
Shield Ethernet et carte Arduino

Voyons de quels composants vous avez besoin.

Composants nécessaires

Ce montage nécessite les composants suivants.

Tableau 21-1 ▶
Liste des composants

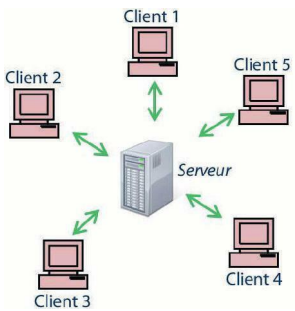
Composant	
1 shield Ethernet	
1 câble réseau	
1 shield d'entrée analogique (ou un potentiomètre distinct)	

ATTENTION AU CABLAGE !

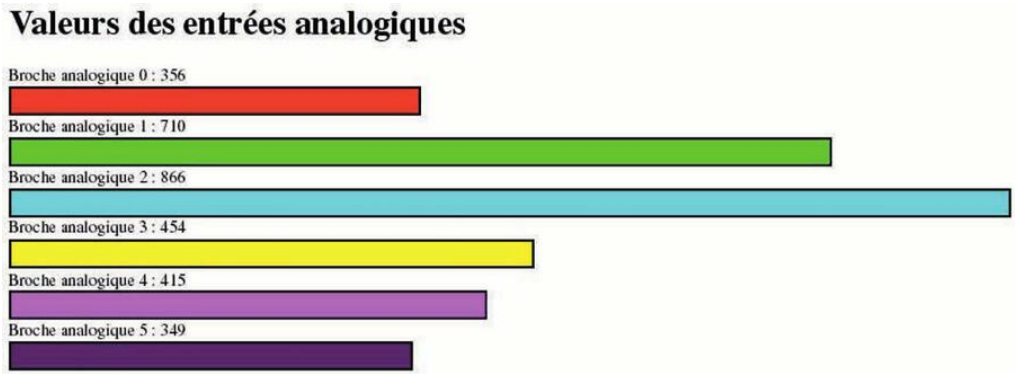
Utilisez un câble normal pour relier votre shield Ethernet à votre routeur. Ces câbles sont jaunes, blancs ou même noirs. Ne vous servez pas d'un câble réseau rouge entre votre routeur et le shield Ethernet, car il s'agit généralement d'un câble croisé qui ne doit être employé que pour relier votre shield directement à la carte réseau de votre ordinateur. Les lignes de réception et d'émission sont alors croisées.

Le sketch suivant permet d'utiliser le shield Ethernet comme un serveur web. Quand vous passez par un navigateur web (tel Firefox, Opera ou IE) pour vous connecter à Internet, vous établissez une liaison avec un serveur web (voir figure 21-5).

Figure 21-5 ▶
Modèle client-serveur



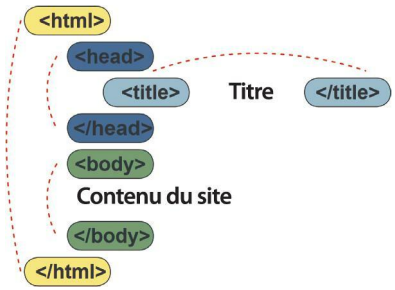
La figure 21-5 montre un serveur (fournisseur) au centre, qui répond aux requêtes de nombreux clients (utilisateurs). Un serveur est un logiciel qui réagit à une demande de contact venant de l'extérieur et délivre des informations. Il peut s'agir d'un serveur mail ou FTP ou encore d'un serveur web. Un client peut être un client mail, tel que Thunderbird ou Outlook. S'il s'agit d'un client web, cela peut être Firefox, Opera ou Chrome, tous déjà mentionnés dans ce livre. Prenons maintenant un exemple concret, dans lequel le shield Ethernet, en tant que serveur web, doit envoyer les valeurs des entrées analogiques de la carte Arduino. La figure 21-6 offre un aperçu de l'affichage dans le navigateur web.



▲ **Figure 21-6**
Affichage de la page HTML dans le navigateur web (représentation numérique et graphique)

Est-ce que cela signifie que vous allez apprendre à programmer un site Internet ? Eh oui, on ne peut pas faire autrement, mais soyez rassuré. Nous n'allons qu'effleurer le sujet, car celui-ci pourrait sans peine remplir toute une bibliothèque. Les sites Internet sont programmés en HTML *(Hypertext Markup Language)*. Il s'agit d'un langage de balisage à base de texte permettant par exemple de représenter du texte, des images, des vidéos ou des liens sur un site Internet que le navigateur web peut lire et afficher. Vous trouverez ci-après la trame de base d'un site, que nous remplirons par la suite pour présenter nos informations. La plupart des éléments HTML sont identifiés par des paires de balises *(tags)*. Il y a toujours une balise de début (ouvrante) et une balise de fin (fermante). La figure 21-7 montre la trame de base en question, les paires correspondantes étant indiquées en couleurs.

Figure 21-7 ▶
Trame de base d'un site Internet



Les lignes pointillées en rouge vous permettent de voir les formations de paires. Les différentes balises ou éléments HTML sont constitués par les noms des éléments entre chevrons. Voyons maintenant une paire de balises de plus près :

Figure 21-8 ▶
La paire de balises `title`

Cette paire génère le titre du site Internet, le texte se trouvant entre la balise de début et la balise de fin. La balise de fin présente le même nom d'élément que la balise de début, cependant il est précédé d'une barre oblique (appelée également slash).

Sketch Arduino

Le sketch initialise un serveur web et fournit des données à un client – c'est-à-dire votre navigateur web.

```
#include <Ethernet.h>

byte MACAddress[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED}; // Adresse MAC
byte IPAddress[] = {192, 168, 178, 200}; // Adresse IP
int const HTTPPORT = 80;                 // Port HTTP 80 (port standard)
String barColor[] = {"ff0000", "00ff00", "0000ff",
                ➥ "ffff00", "ff00ff", "550055"}; // Couleurs RGB pour
                                                 // barres de couleur
#define HTML_TOP    "<html>\n<head><title>Server Web Arduino</title></
➥ head>\n<body>"
#define HTML_BOTTOM "</body>\n</html>"
EthernetServer myServer(HTTPPORT);  // Démarrage du serveur web sur le
                                    // port indiqué

void setup() {
  Ethernet.begin(MACAddress, IPAddress); // Initialisation Ethernet
  myServer.begin();                      // Démarrage du serveur
}
```

```
void loop() {
  EthernetClient myClient = myServer.available();
  if(myClient) {
    myClient.println("HTTP/1.1 200 OK");
    myClient.println("Content-Type: text/html");
    myClient.println();

    myClient.println(HTML_TOP);     // HTML début
    showValues(myClient);           // Contenu HTML
    myClient.println(HTML_BOTTOM);  // HTML fin
  }
  delay(1);        // Courte pause pour navigateur web
  myClient.stop(); // Fermeture connexion client
}

void showValues(EthernetClient &myClient) {
  for(int i = 0; i < 6; i++){
    myClient.print("Analog Pin ");
    myClient.print(i);
    myClient.print(": ");
    myClient.print(analogRead(i));
    myClient.print("<div style=\"height: 15px; background-color: #");
    myClient.print(barColor[i]);
    myClient.print("; width:");
    myClient.print(analogRead(i));
    myClient.println("px; border: 2px solid;\"></div>");
  }
}
```

Pour accéder au serveur web Arduino, écrivez l'adresse IP du code de sketch dans la ligne d'adresse de votre navigateur web Arduino. Dans mon cas, l'adresse est la suivante.

Si cette indication vous semble trop énigmatique, vous pouvez bien sûr choisir une adresse plus parlante :

Il vous suffit d'adapter dans Windows le fichier `hosts` avec des droits d'administrateur sous

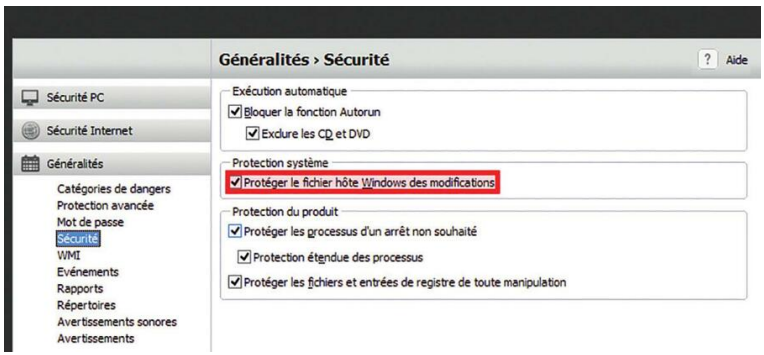
```
C:\Windows\System32\drivers\etc
```

et d'ajouter la ligne dans laquelle j'ai indiqué le nom `Arduino` :

```
# localhost name resolution is handled within DNS itself.
#
127.0.0.1       localhost
#
::1             localhost
192.168.178.200
arduino
```

L'appel est alors plus simple et vous n'avez pas besoin de retenir l'adresse IP. Il arrive parfois qu'un logiciel antivirus interdise l'accès au fichier. Dans Antivir, décochez la case illustrée, corrigez le fichier, puis cochez à nouveau la case.

Figure 21-9 ▶
Paramètres de sécurité pour antivirus



Examinons la signification de ce sketch.

Revue de code

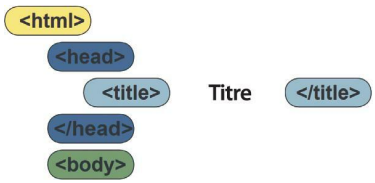
La bibliothèque `Ethernet.h` doit être incorporée pour qu'il soit possible d'utiliser la fonctionnalité du shield Ethernet.

Peut-être croyez-vous qu'il y a une faute de frappe dans la variable `HTTPPORT`, et qu'il s'agit-il de `HTMLPORT` puisqu'il est question ici de pages HTML ? C'est vrai qu'on s'y perd un peu au début. HTTP est la forme abrégée de *HyperText Transfer Protocol*. Comme vous l'avez peut-être remarqué, on a affaire à un grand nombre de protocoles différents en informatique. Quand il s'agit de pages web, ce protocole est chargé de la transmission. Quand vous tapez une adresse web dans votre navigateur, celle-ci commence la plupart du temps par `http://` et non par `html://`. Passons maintenant à la définition du port. Le port standard pour des serveurs web qui utilisent le protocole HTTP est le numéro 80. Imaginez-vous ce numéro comme une sorte de bifurcation sur la route du réseau, où d'autres protocoles circulent encore. Voici une petite liste d'applications dont vous avez peut-être déjà entendu parler.

Port	Service	Objet
21	FTP	Transfert de fichier via FTP-Client
25	SMTP	Envoi d'e-mails
110	POP3	Accès client à un serveur de messagerie

◀ Tableau 21-2
Petite liste de numéros de port et services

Je voudrais encore vous parler brièvement de la structure d'une page HTML. La seule partie variable de notre page est la partie que j'ai appelée *Contenu de ma page*. Tout ce qui est au-dessus ou en dessous ne change pas. C'est pour cette raison que j'ai créé les raccourcis pour la partie supérieure :



dans la définition `HTML_TOP` et pour la partie inférieure :

dans `HTML_BOTTOM`. Vous retrouverez la même chose dans le sketch, avec les lignes suivantes :

```
#define HTML_TOP    "<html>\n<head><title>Server Web Arduino</title></head>\
n<body>"
#define HTML_BOTTOM "</body>\n</html>"
```

La séquence d'échappement `\n` provoque un saut de ligne, de telle sorte que le code HTML soit formaté d'une certaine manière et que tout ne soit pas mis sur une seule ligne. Venons-en maintenant au déroulement du sketch proprement dit. Diverses parties du programme sont, comme toujours, initialisées dans la fonction `setup`.

```
void setup() {
  Ethernet.begin(MACAddress, IPAddress);  // Initialisation Ethernet
  myServer.begin();                       // Démarrage du serveur
}
```

La première étape consiste à doter le shield Ethernet de l'adresse MAC et d'une adresse IP unique.

Vous vous demandez certainement d'où sort l'adresse IP 192.168.2.110 en question. Eh bien, c'est tout simple ! Mon routeur se trouve dans la zone d'adresse `192.168.2` et l'adresse d'hôte `1` lui est attribuée, autrement dit son adresse IP est `192.168.2.1`. Je peux donc affecter des adresses comprises entre `192.168.2.2` et `192.168.2.254` à d'autres utilisateurs du réseau. Revenons à l'initialisation. La deuxième étape consiste à démarrer le serveur web, de sorte qu'il puisse réagir à des demandes entrantes. Celui-ci épie le réseau et reste sur le qui-vive jusqu'à ce qu'un client l'aborde et lui demande quelque chose. Il accomplit ensuite son travail et délivre les données avant de se remettre à nouveau en position d'attente. Passons maintenant au traitement proprement dit dans la fonction `loop`. La présence de la demande d'un client est d'abord vérifiée :

```
EthernetClient myClient = myServer.available();
if(myClient){...}
```

Si l'interrogation `if` est satisfaite, le serveur peut commencer à envoyer ses informations au client.

Vous remarquerez que cette interrogation n'est que la forme abrégée du code suivant :

```
if(myClient == true){...}
```

L'interrogation sur `true` est facultative du fait que si l'expression est vraie dans l'instruction `if`, c'est le bloc subséquent qui est exécuté. Vous n'avez pas besoin de vérifier à nouveau avec `==` que l'expression est vraie. Vous comprenez maintenant ? Donc, si un client a effectué une demande auprès du serveur, ce dernier renvoie pour commencer les lignes suivantes :

```
myClient.println("HTTP/1.1 200 OK");
myClient.println("Content-Type: text/html");
myClient.println();
```

Dans la première ligne, le serveur confirme la demande du client en transmettant la version `1.1` du protocole HTTP, suivie du code d'état `200` indiquant que la demande a été traitée avec succès et que le résultat de la demande est transmis dans la réponse. La deuxième ligne indique ce que l'on apelle le *type MIME*, à savoir le genre des données envoyées par le serveur (`text/html` dans notre cas). S'agit-il d'informations purement textuelles ou une image est-elle éventuellement délivrée au client ? Les données transmises doivent alors être bien sûr interprétées en conséquence et non pas affichées en texte clair. Passons maintenant au code, qui envoie les données lues de votre carte Arduino :

```
myClient.println(HTML_TOP);    // HTML début
showValues(myClient);          // Contenu HTML
myClient.println(HTML_BOTTOM); // HTML fin
```

Les tâches de `HTML_TOP` et `HTML_BOTTOM` vous sont connues. L'appel des données de la carte est exécuté par la fonction `showValues` que je vous redonne ici :

```
void showValues(EthernetClient &myClient) {
  for(int i = 0; i < 6; i++){
    myClient.print("Analog Pin ");
    myClient.print(i);
    myClient.print(": ");
    myClient.print(analogRead(i));
    myClient.print("<div style=\"height: 25px; background-color: #");
    myClient.print(barColor[i]);
    myClient.print("; width:");
    myClient.print(analogRead(i));
    myClient.println("px; border: 2px solid;\"></div>");
  }
}
```

Vous noterez que dans l'en-tête de fonction , il y a un « et » commercial (`&`) devant le paramètre `myClient`. Ce signe distinctif est en fait une *référence*. Quand on passe une variable comme paramètre d'une fonction, celle-ci travaille avec une copie de cette variable, ce qui n'a aucune influence sur la variable d'origine. La fonction peut par exemple doubler la valeur du paramètre. L'originale demeure inchangée. Mais pour pouvoir utiliser l'objet `Client` original dans la fonction, l'adresse mémoire de l'original est communiquée au moyen de l'opérateur de référence `&`. Dans la fonction, je travaille quasiment avec l'original. La fonction affiche d'une part les valeurs des entrées analogiques et de l'autre des barres horizontales. J'utilise pour ce faire la balise `div`, qui peut servir de contenant pour d'autres éléments HTML. Je m'en sers ici pour colorer une certaine zone. Il est possible de donner des informations de hauteur ou de largeur au moyen d'une indication `style`. Une ligne HTML peut alors ressembler à cela :

```
Analog Pin 0: 168<div style="height: 25px; background-color: #ff0000;
width:168px; border: 2px solid;"></div>
```

La zone `div` a ici une hauteur de 25 pixels et une largeur de 168 pixels. Vous trouverez des informations détaillées dans la littérature spéciale ou sur Internet.

POUR ALLER PLUS LOIN

Pour compléter ce chapitre, vous pouvez effectuer une recherche sur Internet sur les mots-clés :
- selfhtml ;
- cascading stylesheets ;
- div-tag.

Peut-être avez-vous constaté, lors de votre réalisation, que les valeurs des entrées analogiques s'affichent un point c'est tout. Si vous tournez l'un des potentiomètres, rien ne bouge sur la page Internet. C'est normal car le navigateur web appelle une page auprès du serveur web et en assure la présentation (ce procédé est également appelé *rendu*). Si le navigateur n'émet aucune autre demande, le contenu de la page demeure bien entendu inchangé. Vous pouvez toujours appuyer assez souvent sur la touche Actualiser (F5). Mais je doute que cela vous donne satisfaction. Modifiez plutôt dans votre sketch la ligne de code où `HTML_TOP` a été défini et vous verrez que le comportement de votre navigateur changera.

```
#define HTML_TOP "<html>\n<head><title>Server Web Arduino</title></
head>\n \
<meta http-equiv=\"refresh\" content=\"1\">\n<body>"
```

Le passage suivant est décisif :

```
<meta http-equiv=\"refresh\" content=\"1\">
```

La balise `meta` en question demande au navigateur d'exécuter automatiquement une actualisation (`refresh`) toutes les secondes. Le backslash \ à la fin de la première ligne définissant `HTML_TOP` permet que cette ligne se poursuive sur la suivante. Faute de quoi une erreur de compilateur se produirait.

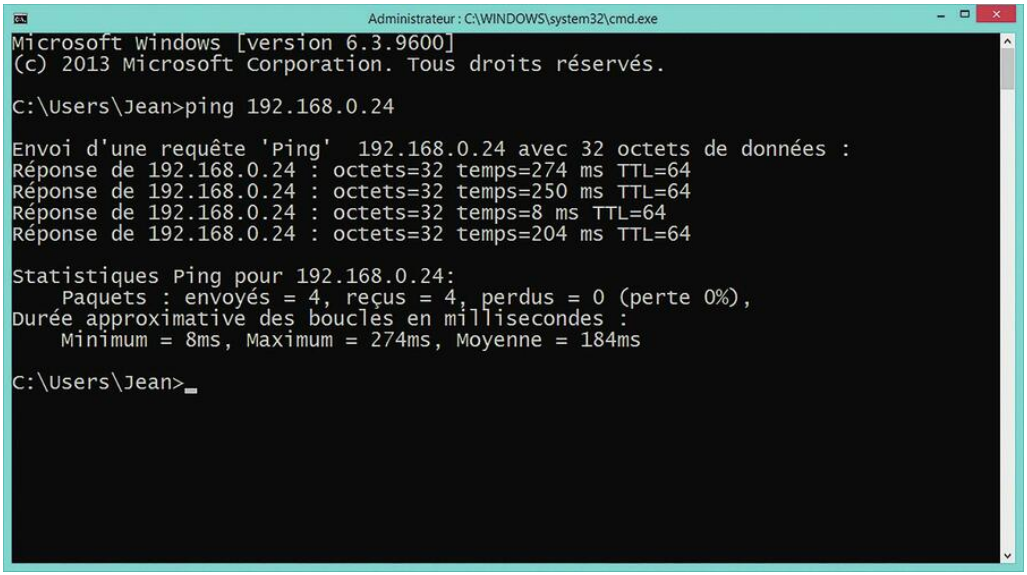
Exercice complémentaire

Écrire un nouveau sketch montrant, en plus des entrées analogiques, l'état des entrées numériques sur votre page web Arduino.

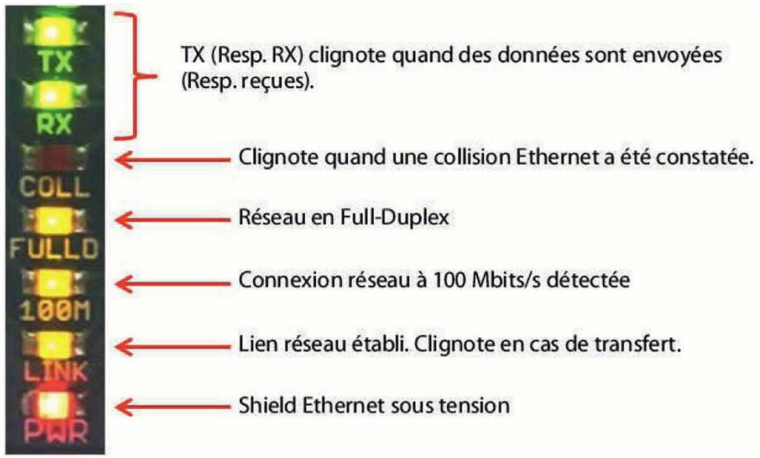
Problèmes courants

Vérifiez ce qui suit si la page du serveur web ne s'affiche pas.

- Avez-vous saisi la bonne adresse IP dans la ligne d'adresse de votre navigateur ? Elle doit correspondre à celle de votre sketch.
- Pouvez-vous atteindre le serveur web en tapant une commande `ping` dans la ligne de commande ? Sinon, vérifiez votre câble réseau ou éventuellement les réglages du pare-feu. Une exécution réussie de la commande `ping` donne le résultat suivant.



Le shield Ethernet possède certaines LED qui donnent des informations sur l'état (voir figure page suivante).



Vérifiez l'affichage des LED. Les LED PWR et LINK doivent être allumées. La LED 100M ne s'allume que dans le cas d'un réseau 100 Mbits/s. Elle reste éteinte pour 10 Mbits/s. Si des données sont envoyées toutes les secondes comme dans le dernier exemple, les LED TX et RX clignotent au même rythme.

Qu'avez-vous appris ?

- Vous avez appris à réaliser un serveur web avec le shield Ethernet.
- Vous avez interrogé les entrées analogiques et vu comment s'affichent les valeurs à peu de chose près en temps réel.
- La trame de base d'une page HTML devrait maintenant vous être plus familière.
- Vous maîtrisez les notions de base relatives aux réseaux.

Montage

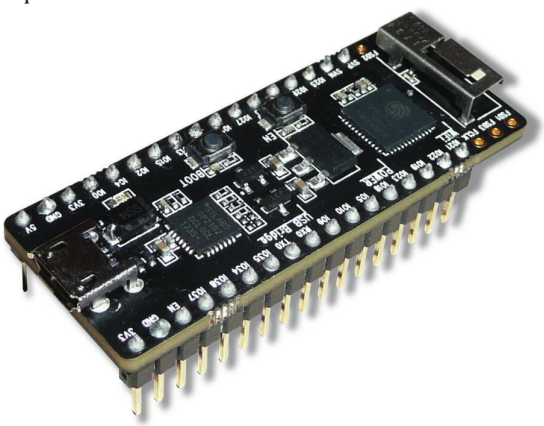
22

La carte ESP32

L'Internet des objets (*Internet of Things* en anglais, ou IoT) et ses applications sont aujourd'hui la norme et occupent une place de plus en plus importante dans l'univers des développeurs amateurs. Ce qui était autrefois réalisé par des architectures client-serveur prend maintenant une tout autre allure dans l'environnement IoT. Il a fallu faire évoluer les bus physiques spécifiques aux applications pour les adapter aux interfaces de réseau ouvertes telles que les réseaux locaux (LAN et Wi-Fi) ou Bluetooth. Dans ce montage, vous allez découvrir un autre microcontrôleur, l'ESP32. L'ESP32 est un microcontrôleur 32 bits bon marché et de faible puissance, qui intègre la connectivité sans fil (Wi-Fi). L'intérêt de ce petit composant est qu'on peut le programmer avec l'environnement de développement Arduino.

Présentation de la carte ESP32

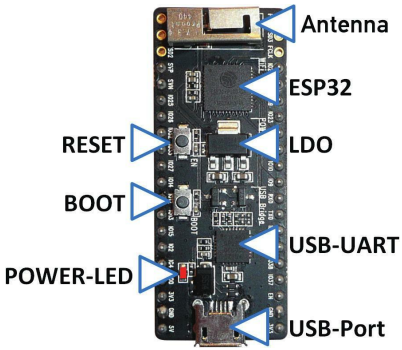
Le module ESP32 est un microcontrôleur développé par la société chinoise Espressif. Cette puce constitue le perfectionnement logique du module ESP8266. Plus de mémoire SRAM, une vitesse de processeur plus élevée, une communication Bluetooth, un plus grand nombre de ports (GPIO), des capteurs tactiles, une conversion A/N et N/A sont quelques-uns des points forts de l'ESP32, si bien que l'on rencontre de plus en plus son module monté sur des cartes. Une carte de développeur avec une puce ESP32 est représentée ci-dessous :



◀ **Figure 22-1**
L'ESP32-Pico-Board V4

Il s'agit ici de l'ESP32-Pico-Board V4. Jetons un œil sur ce module et voyons de quels composants cette petite platine est constituée. Cette carte est en vente partout à moins de 10 euros. La figure suivante montre les principaux composants qui nous intéressent pour le moment :

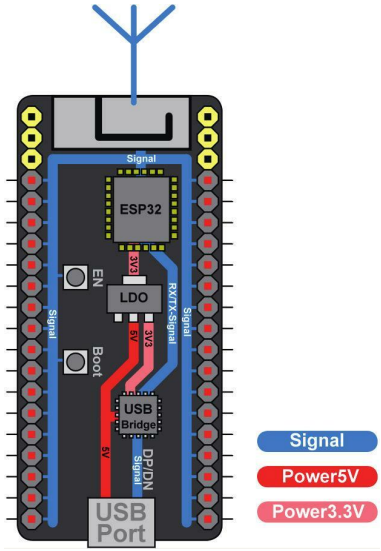
Figure 22-2 ▶
L'ESP32-Pico-Board V4 et ses composants



Quelles sont les fonctions des composants indiqués ci-dessus ?

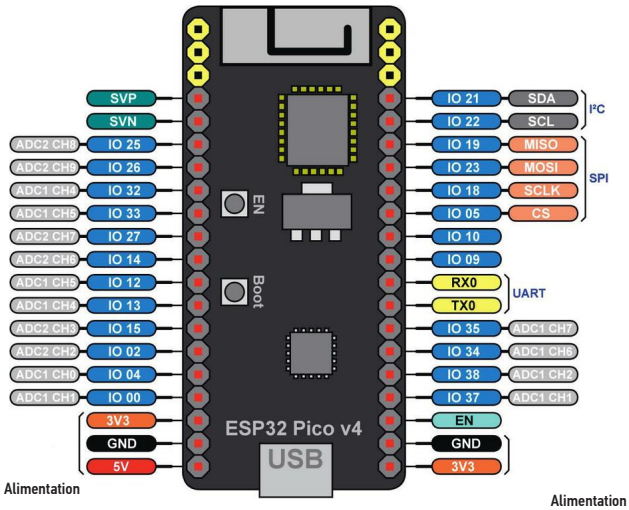
- *Reset* : bouton-poussoir miniature portant le marquage *EN* qui permet de redémarrer le système lorsqu'on appuie dessus. Il existe également une broche avec la même désignation permettant de déclencher un Reset par contact à la masse (LOW actif).
- *Boot* : autre bouton-poussoir miniature portant le marquage *BOOT*, pouvant être utilisé en association avec le bouton *EN* pour charger un firmware. Il ne nous est toutefois pas utile pour le moment, car le téléchargement de firmware, qui sera ensuite enregistré sur la carte, sera effectué par l'environnement de développement Arduino que nous utiliserons.
- *Power-LED* : petite LED qui s'allume lorsque la carte est alimentée en tension par un port USB ou via une source externe.
- *Antenna* : nécessaire pour la communication sans fil.
- *ESP32* : microcontrôleur ESP32-PICO-D4.
- *LDO* : étant donné que la carte ESP32 fonctionne sous une tension de 3,3 V, la tension de 5 V issue du port USB doit être abaissée et régulée à 3,3 V par un régulateur de tension LDO *(Low Dropout Voltage Regulator)*.
- *USB-UART* : il s'agit d'une puce CP1202 de Silicon Labs qui crée un pont entre l'USB et l'UART (dispositif de communication série), assurant la transmission de données. Le débit de transfert est d'1 Mbps.
- *USB-Port* : ici une micro prise femelle USB 2.0 pour connecter la carte à l'ordinateur.

Les liaisons entre les différents composants sont visibles sur le schéma-bloc fonctionnel suivant :



◀ Figure 22-3
Schéma fonctionnel en blocs de l'ESP32-Pico-Board V4

Examinons maintenant les différentes broches de l'ESP-32-Pico-Board ; de nombreuses informations sont à prendre en compte. La figure suivante montre les broches les plus importantes et leurs fonctions, comme l'alimentation en tension, IO (Entrées-Sorties), ADC (Conversion analogique-Numérique), le bus I²C et SPI (Bus série synchrone) :



◀ Figure 22-4
L'ESP32-Pico-Board V4 et ses broches

Nous voyons que certaines broches ont deux fonctions pouvant être activées selon la programmation. Il existe des fonctions primaires et des fonc-

tions alternatives. Le nombre de broches a ainsi été réduit sur la carte pour diminuer sa taille autant que possible. Afin que les informations ne soient pas trop déroutantes, surtout pour les débutants, je n'ai représenté sur la figure que les détails qui sont les plus importants pour nous à ce stade. Un bon nombre de ces broches présentent d'autres fonctionnalités qui sont secondaires pour nous. Pour en savoir plus sur le sujet, vous pouvez consulter le site Internet suivant :

https://esp-idf.readthedocs.io/en/latest/get-started/get-started-pico-kit.html

LES TENSIONS SUR LA CARTE ESP32

Il faut bien noter que la carte, le microcontrôleur ESP32, fonctionne avec une tension de service de 3,3 V. Tout ce qui est supérieur à cette tension entraîne la destruction de la carte. La tension de service de 5 V, courante sur la carte Arduino Uno, ne doit pas être utilisée.

Je voudrais ici présenter quelques caractéristiques importantes de l'ESP32-Pico-Board, qui intéresseront certainement quelques-uns d'entre vous. La liste n'est pas exhaustive. Pour des informations détaillées, je vous renvoie au site du fabricant Espressif.

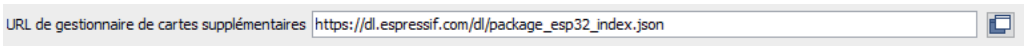
Tableau 22-1 ▶ Spécifications de l'ESP32-Pico-Board

Spécification détaillée	
Processeur	32-Bit Dual Core (deux microprocesseurs LX6 32 bits Low-Power Xtensa)
Mémoire interne	• Mémoire ROM 448 Ko pour l'amorçage et les fonctions Core • Mémoire SRAM 520 Ko (8 Ko RTC FAST Memory) (On-Chip) pour les données et les instructions • Mémoire SRAM 8 Ko dans RTC (RTC FAST Memory) pour les données et l'utilisation par le processeur pendant le processus d'amorçage RTC à partir du mode Deep-Sleep • Mémoire SRAM 8 Ko dans RTC (RTC SLOW Memory) pour l'accès du coprocesseur pendant le mode Deep-Sleep • eFuse 1 Ko, dont 320 octets sont utilisés pour l'adresse MAC et la configuration de la puce. Les 704 octets restants peuvent servir aux fonctions propres comme Flash Encryption ou l'ID de la puce.
Mémoire externe	• Accès à la mémoire QSPI Flash et à la mémoire SRAM via caches High-Speed • Mémoire Flash Memory-Mapped externe de 16 Mo maximum dans la plage de code du processeur. Compatible avec un accès 8, 16 ou 32 bits. • Mémoire Flash/SRAM Memory-Mapped externe de 8 Mo maximum dans la plage de données du processeur. Compatible avec un accès 8, 16 ou 32 bits. Data-Read compatible pour mémoire Flash et SRAM, Data-Write proposé seulement pour mémoire SRAM. • Mémoire SPI Flash externe 4 Mo. Mémoire SPI Flash Memory-Mapped 4 Mo dans la plage de code du processeur. Compatible avec un accès 8, 16 ou 32 bits et Code Execution.

Spécification détaillée	
Fréquence	Oscillateur quartz 40 MHz
Wi-Fi	2,4 GHz HT40
Bluetooth	BLE 4.2 (Bluetooth-Low-Energy)
Périphérie	SPI, I²C, I²S, UART, CAN 2.0 et interface Ethernet ADC 12 bits (convertisseur analogique-numérique)
Capteurs	Tactile, Hall et température
MLI	1 canal matériel et 16 canaux logiciels MLI (modulation de largeur d'impulsion)
ES	Broches GPIO (General Purpose Input/Output)

L'intégration de l'ESP32 et un premier test

Pour intégrer les fonctions du module ESP32 dans l'environnement de développement Arduino, sélectionnez dans l'IDE Arduino l'option de menu *Fichier* | *Préréglages* pour accéder à la boîte de dialogue de configuration et entrez l'URL suivante :



Sélectionnez ensuite l'option de menu *Outils* | *Type de carte* | *Gestionnaire de carte* pour ouvrir le gestionnaire de carte éponyme. Cherchez *ESP32* et installez la bibliothèque en cliquant sur le bouton *Installer*.

▲ **Figure 22-5**
Installation de la compatibilité ESP32

Cliquez sur l'option de menu *Outils* | *Type de carte* pour trouver une nouvelle entrée portant le nom *ESP32 Boards*. Vous pouvez voir toutes les cartes compatibles et sélectionner celle que vous recherchez (figure 22-6).

La liste est trop longue pour être représentée ici. L'entrée qui nous importe est *ESP32 Pico Kit*. Si vous avez opté pour une autre carte, vous devrez la sélectionner dans la liste. Mais avant de l'acheter, vérifiez qu'elle figure bien dans cette liste.

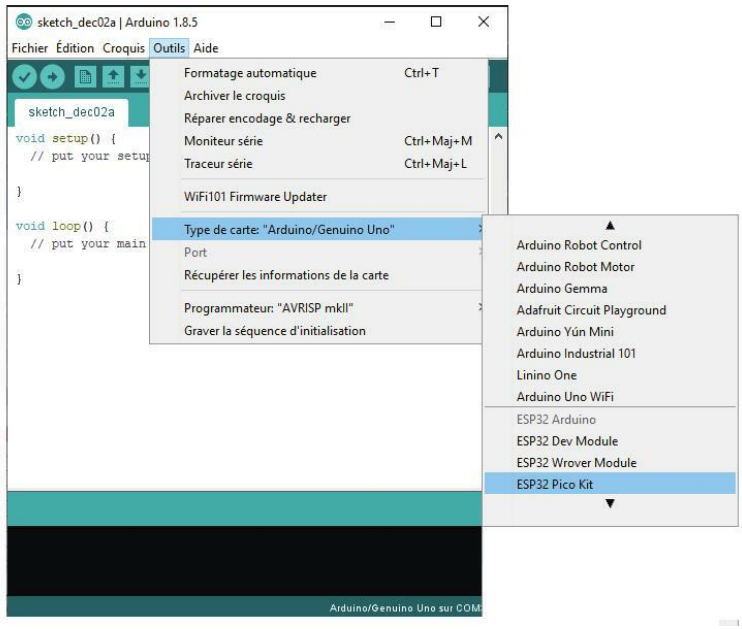
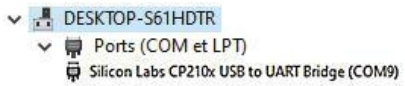


Figure 22-6 ▶
Sélection de votre carte ESP32

En règle générale, le pilote correspondant est installé automatiquement par Windows 10 :

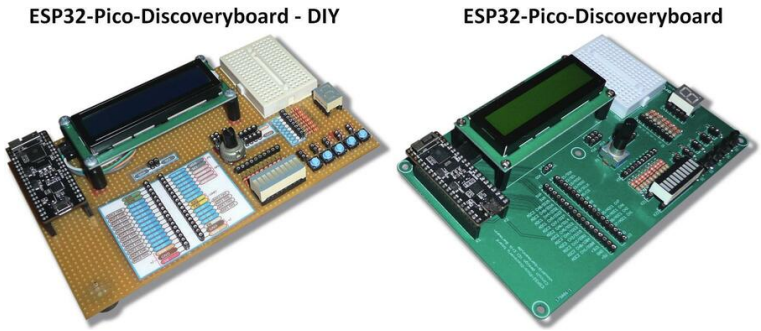


Une fois l'installation réussie, vous pouvez vous livrer à un premier test. C'est ce que nous allons faire avec un test *LED-Hello-World*, qui ne devrait pas prendre trop de temps. Mais avant de commencer, je vais vous montrer, parallèlement à l'installation sur une plaque de prototypage ordinaire, deux solutions intéressantes pour réaliser la mise en circuit.

L'ESP32-Pico-Discoveryboard

La figure suivante montre deux modèles d'une même carte, que j'ai nommée ESP32-Pico-Discoveryboard : à gauche, un modèle basé sur une carte au format Europe 160 ×100 mm, qui a donc été soudé par vos soins, et à droite une carte réalisée par un fabricant professionnel.

Vous connaissez déjà mon penchant pour les cartes faites maison. C'est pourquoi je vous montre ces deux solutions.



◀ **Figure 22-7**
Les ESP32-Pico-Discoveryboards

L'ESP32-Pico-Discoveryboard dispose déjà des divers éléments de base, électriques et électroniques, parmi lesquels :

- un support pour l'ESP32-Pico-Board ;
- des gabarits de broche ESP32 (à enregistrer avec des couleurs de votre choix) avec des prises femelles pour une orientation et un câblage plus faciles ;
- 10 LED dotées de résistances en série adéquates et de prises femelles ;
- 5 boutons-poussoirs miniatures avec résistances de rappel (Pull-down) correspondantes et prises femelles ;
- un potentiomètre ;
- un affichage sept segments doté de résistances en série adéquates et de prises femelles ;
- un écran LC piloté par le bus I^2C et des prises femelles ;
- des rails d'alimentation pour 3,3 V et GND ;
- une petite plaque de prototypage (Breadboard) pour loger des composants électroniques supplémentaires.

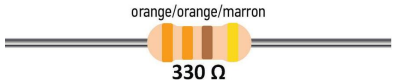
Vous pouvez aussi prendre une plaque de prototypage tout à fait normale et y connecter les composants en fonction de vos centres d'intérêt. Si vous souhaitez ne tester qu'une fois l'ESP32-Board pour connaître ses capacités, je vous conseille de prendre une plaque de prototypage. Et si cela vous convient mieux, vous pouvez l'utiliser au lieu d'une des deux Discoveryboards. Le principal avantage est que toutes les broches de l'ESP32 sont marquées et que, comme la Discoveryboard Arduino, certains composants de base sont fournis d'office. Vous trouverez représentée ci-après la mise en circuit sur l'ESP32-Pico-Discoveryboard.

Faire clignoter avec le module ESP32

Pour communiquer avec le monde extérieur, la carte de développeur représentée dispose, comme on l'a vu, de broches IO, IO étant l'abréviation de *Input/Output* (Entrée/Sortie en français). Des informations peuvent être fournies à la carte et envoyées par ces broches, tout comme sur la carte Arduino Uno.

Pour ce montage, nous aurons besoin des composants suivants :

Tableau 22-2 ▶ Liste des composants

Composant	
Carte ESP32	
1 LED rouge	
1 résistance de 330 Ω	
1 résistance de 10 kΩ	
1 bouton-poussoir miniature	

La figure 22-8 montre à l'aide du schéma des connexions comment réaliser les deux sens du flux de données.

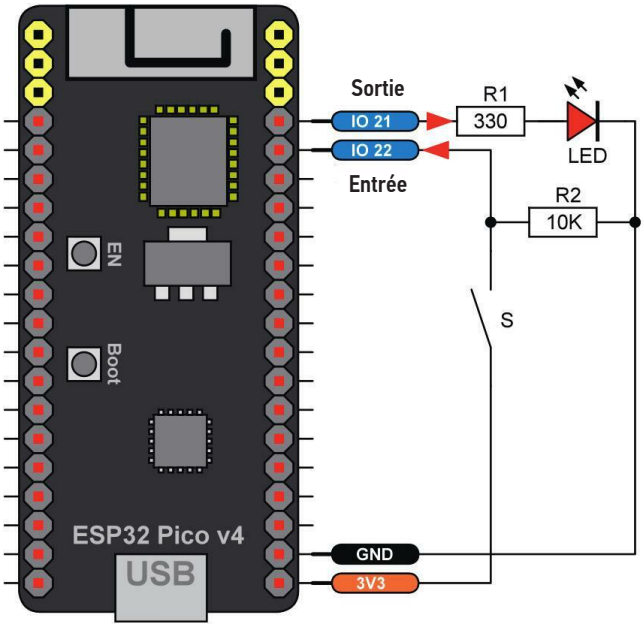


Figure 22-8
Exemple simple d'activités IO

La broche portant la désignation *IO 21* est utilisée comme sortie et pilote via une résistance en série, une diode électroluminescente ou LED. Dans le sens opposé, c'est la broche portant la désignation *IO 22* qui fait office d'entrée pour interroger un bouton et la résistance de rappel (Pull-down) qui est utilisée. La carte ESP32-Pico-Discoveryboard possède déjà des résistances de rappel (Pull-down) à ses six boutons, bien que celles-ci ne soient pas nécessaires (comme sur la carte Arduino Uno), car il existe des résistances de tirage (Pull-up) internes dans l'ESP32.

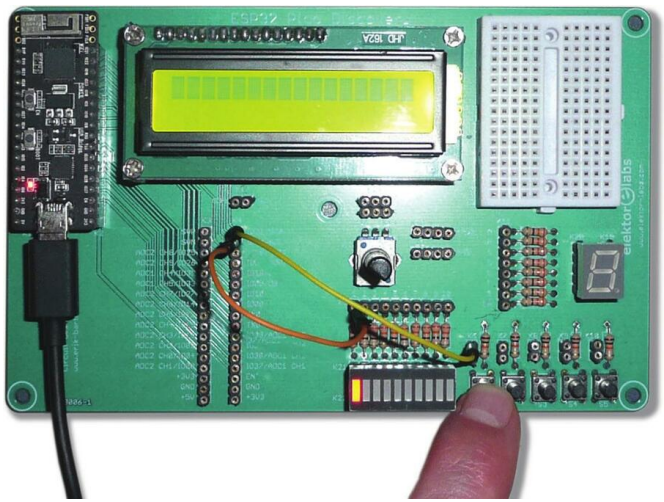


Figure 22-9
Circuit sur l'ESP32-Pico-Discoveryboard

Le code ESP32 pour interroger le bouton et commander la LED est :

```
define bouton 21 // Broche bouton
#define LED 22   // Broche LED

void setup() {
pinMode(bouton, INPUT);
pinMode(LED,    OUTPUT);
}

void loop() {
digitalWrite(LED, digitalRead(bouton));
}
```

Une fois le code chargé sur la carte ESP32, le message dans l'environnement de développement Arduino est un peu différent de celui auquel on est habitué sur Arduino Uno. Par manque de place, je n'ai pas représenté ici toutes les lignes d'édition, mais seulement un petit extrait. La dernière ligne est importante. Elle indique un reset du module ESP32, qui vous laisse présager que le chargement a fonctionné.

```
  esptool.py v2.6
  Serial port COM9
  Connecting.....
  Chip is ESP32-PICO-D4 (revision 1)
  Features: Wi-Fi, BT, Dual Core, Embedded Flash, Coding Scheme None
MAC: d8:a0:1d:40:9e:e8
  Changing baud rate to 921600
  Changed.
  Configuring flash size...
  Auto-detected Flash size: 4MB
  Compressed 8192 bytes to 47...
  Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.0 seconds
(effective 10922.6 kbit/s)...
  Leaving...
  Hard resetting via RTS pin...
```

L'ESP32-Board D1 R32

Il existe bien sûr d'autres cartes de développeur ESP32, proposées par différentes sociétés. Ces cartes présentent une disposition de broches différente et certaines ont un plus grand nombre de broches. Les montages indiqués peuvent certainement être réalisés avec d'autres cartes ESP32. Il faudra cependant veiller au *brochage* respectif, c'est-à-dire à la disposition des broches. Vous la trouverez sur Internet. La figure 22-10 page suivante montre une carte qui ressemble à une carte Arduino. Il s'agit de la carte *D1 R32* de la marque AZ-Delivery. Elle a le même format que la carte Arduino Uno et les en-têtes déjà bien connus.

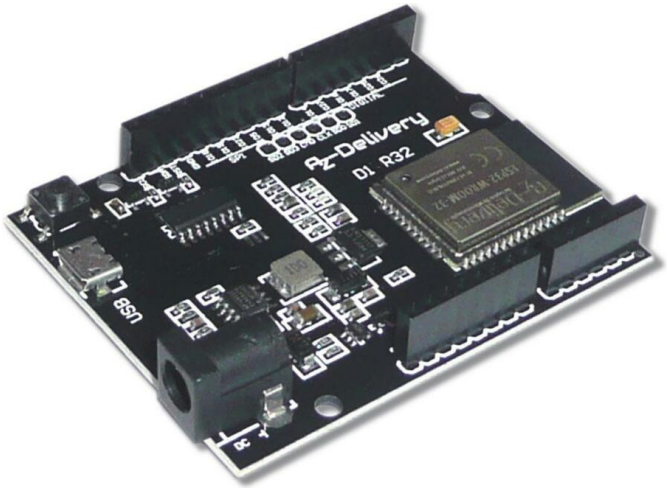


Figure 22-10
La carte D1 R32 de AZ-Delivery

Montage : le log de températures

Voilà pour les informations préalables. Je voudrais vous présenter un montage intéressant, qui affiche non seulement une valeur de mesure localement, mais qui lui permet aussi d'utiliser le réseau Internet afin de pouvoir y être représentée dans un graphique. Mon objectif est de développer avec vous un sketch (je ne suis pas sûr que le terme sketch soit approprié pour une carte ESP32, mais tant pis !). Le sketch qui suit doit établir une connexion à votre routeur par Wi-Fi. En Allemagne, on emploie communément le terme *WLAN* pour désigner un réseau sans fil alors que dans d'autres pays, on utilise généralement le terme Wi-Fi. Il y a certes de petites différences, mais elles ne nous importent pas ici. Avant de commencer, regardez d'abord les données d'accès Wi-Fi à votre routeur, car vous en aurez besoin pour connecter un appareil à votre réseau domestique. Au début du sketch, vous devez signaler à votre ESP32 que vous souhaitez utiliser la fonctionnalité Wi-Fi. Une instruction `include` correspondante supplémentaire est donc nécessaire. Supposons que les données d'accès dans un réseau domestique soient les suivantes :

- SSID : *EriksWLan* ;
- mot de passe : *Schnickschnack*.

Le sketch commencerait ainsi :

```
#include <WiFi.h>
const char* ssid     = "EriksWLan";
const char* password = "Schnickschnack";
```

Vous avez posé la première pierre à l'établissement de la connexion. À l'intérieur de la fonction `setup`, l'interface sérielle pour le contrôle de la liaison Wi-Fi est initialisée à une vitesse de 115 200 bauds et l'objet `WiFi` est initialisé avec les données d'accès. Dans la boucle `while` a lieu une interrogation continue de l'état de la connexion, qui passe à `WL_CONNECTED` lorsque la liaison Wi-Fi a été établie. Tant que cela n'est pas le cas, la boucle `while` n'est pas interrompue. Une courte pause de 500 ms est ajoutée et un point est édité pour le contrôle visuel à chaque itération de la boucle.

```
void setup() { Serial.begin(115200);
  WiFi.begin(ssid, password);
  Serial.println("Connection au réseau Wi-Fi...");
  while(WiFi.status() != WL_CONNECTED) {
    delay(500);          // Courte pause
    Serial.print(".");  // Editer point si pas encore de connexion
  }
  Serial.println("\nConnecté au réseau Wi-Fi"); Serial.print("Adresse
IP: ");
  Serial.println(WiFi.localIP());
}

void loop() {/* ... */}
```

En absence de connexion au routeur si les données d'accès sont incorrectes, le résultat dans le moniteur série de l'IDE Arduino s'affiche comme indiqué ci-après.

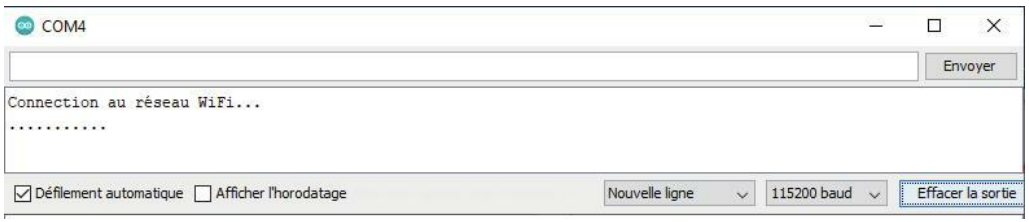
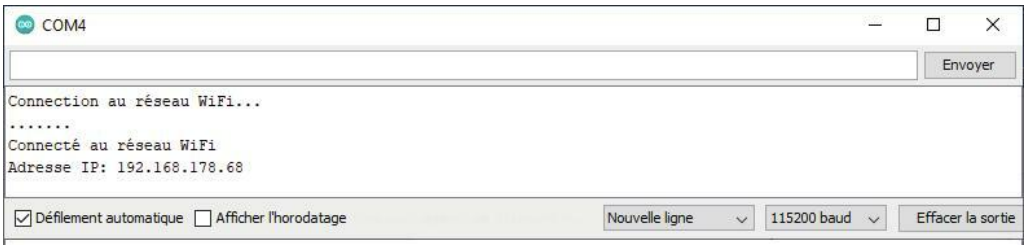


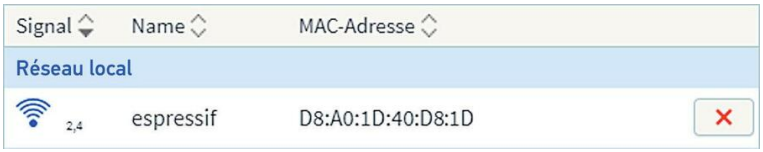
Figure 22-11 ▲
Pas encore de connexion entre l'ESP32 et le routeur

La série de points devenant de plus en plus longue, il faut parfois appuyer brièvement sur le bouton Reset du module ESP32. En revanche, si la connexion Wi-Fi fonctionne plus ou moins directement, l'affichage dans le moniteur série est le suivant et l'adresse IP attribuée par le routeur s'affiche elle aussi :



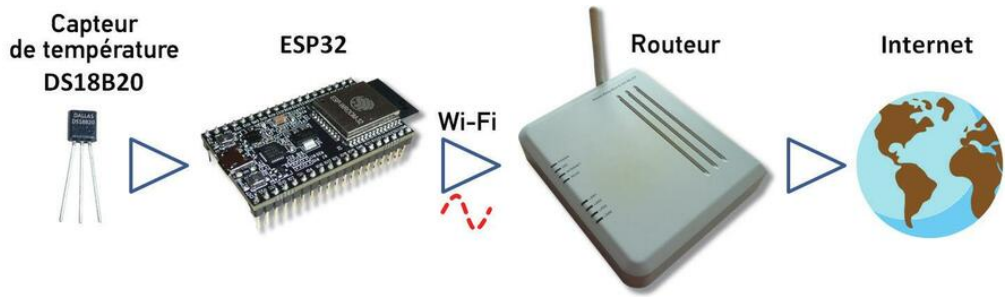
▲ **Figure 22-12**
Connexion établie entre l'ESP32 et le routeur

Voici donc les conditions préalables à la suite de la procédure et à l'extension du sketch, car nous souhaitons obtenir une sorte de trafic par Wi-Fi. Mon routeur Fritzbox a détecté le nouvel appareil sous la forme de la carte ESP32 :



Le serveur ThingSpeak

Pour que le processus visé de collecte et de transfert de données sur Internet fonctionne, il nous faut une contrepartie correspondante sur Internet, qui fasse office de serveur et qui puisse établir une liaison avec mon client local. Comme le laisse entendre le nom du montage n° 13 « La température », le but est de collecter des valeurs de température. Le schéma suivant vous indique clairement le chemin que nous prévoyons entre les instances :



▲ **Figure 22-13**
Parcours de l'information température à travers les différentes instances jusqu'à Internet

Pour ce montage, nous aurons besoin du composant suivant :

Tableau 22-3 ▶
Liste des composants

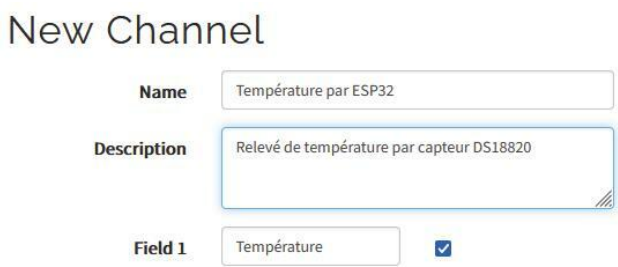
Composant	
Capteur de température (thermistance) DS18B20	

Pour notre montage, nous utiliserons *ThingSpeak,* un service cloud gratuit. Avant de pouvoir l'utiliser, vous devez vous y inscrire gratuitement sur le site Internet du fournisseur :

https://thingspeak.com/

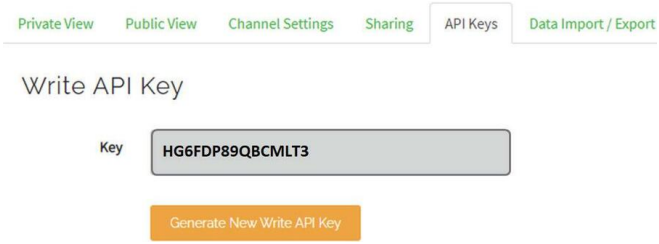
Inscrivez-vous (je ne montre pas l'inscription ici) en confirmant l'adresse e-mail que vous avez indiquée. Vous pourrez ensuite commencer. Une fois inscrit, vous devez créer un *Channel* (une chaîne, en français), dans lequel seront traitées, mais aussi affichées, les données que vous allez envoyer à ce serveur. Allez dans le menu *Channels* puis cliquez sur le bouton *New Channel* :

J'ai pour ma part entré quelques informations pour décrire la chaîne et je les ai confirmées en cliquant sur le bouton vert *Save* en bas de la page :



Nous obtenons la vue suivante :

Nous y voyons un diagramme avec les métadonnées que j'ai indiquées, qui ne contiennent pas encore de données, ce qui est logique puisque rien n'a encore été envoyé au serveur. Pour pouvoir le faire, vous devez générer une clé qu'il est possible de créer dans l'onglet *API Keys* :



◀ **Figure 22-14**
Clé API personnelle pour l'écriture des valeurs de mesure sur la chaîne ThingSpeak

Cette clé est indispensable pour envoyer les valeurs de mesure sur le serveur ThingSpeak. Pour l'envoi d'une seule valeur mesurée à l'aide du navigateur, il faut utiliser le mode d'écriture suivant :

Vous devez saisir ici votre propre clé API. La description du champ se trouve sous l'onglet *Channel Settings*. Après la description du champ, vous devez saisir dans l'URL un signe = suivi de la valeur correspondante. Lorsque le transfert a réussi, le serveur répond par un chiffre qui correspond au nombre de valeurs de mesure envoyées. Dans mon navigateur, j'ai entré successivement les lignes suivantes :

```
https://api.thingspeak.com/update?api_key=J7HKMNNSVU2QH6TK&field1=17
```

```
https://api.thingspeak.com/update?api_key=J7HKMNNSVU2QH6TK&field1=21
https://api.thingspeak.com/update?api_key=J7HKMNNSVU2QH6TK&field1=34
https://api.thingspeak.com/update?api_key=J7HKMNNSVU2QH6TK&field1=4
```

Veillez à toujours maintenir une petite pause d'au moins dix secondes entre les différentes saisies et de ne pas insérer d'espace vide dans la chaîne de caractères. Cela m'est arrivé plusieurs fois et je me suis demandé pourquoi rien ne se passait. Le résultat dans ma chaîne se présente sous la forme suivante :

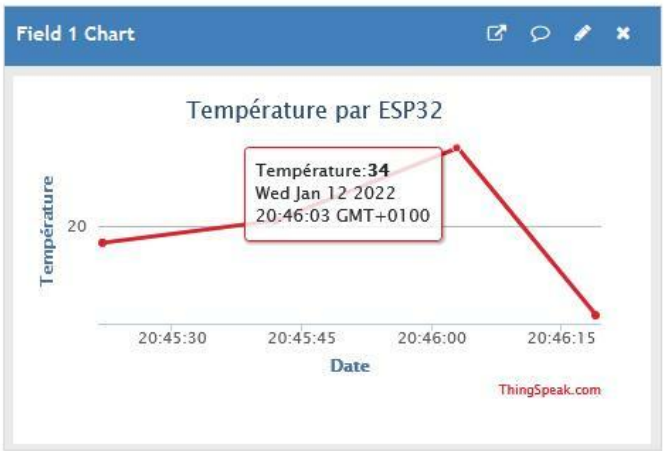


Figure 22-15 ▶
Ma chaîne de températures avec les valeurs envoyées

Si vous déplacez le pointeur de la souris sur l'un des points de mesure rouges, l'infobulle vous donnera des informations supplémentaires sur la valeur exacte ainsi que la date et l'heure. Vous avez vu comment s'effectue le transfert des valeurs via le navigateur, mais c'est le module ESP32 qui doit s'en charger avec la carte Arduino Uno. C'est en principe le même procédé, sauf que la transmission des données s'opère avec une méthode de requête appelée *HTTP-POST*. Cette requête, qui est envoyée au serveur ThingSpeak, contient de multiples informations standards, comme un en-tête, la clé API avec la description du champ, ainsi que les valeurs mesurées, par exemple :

```
J7HKMNNSVU2QH6TK&field1=24.63
```

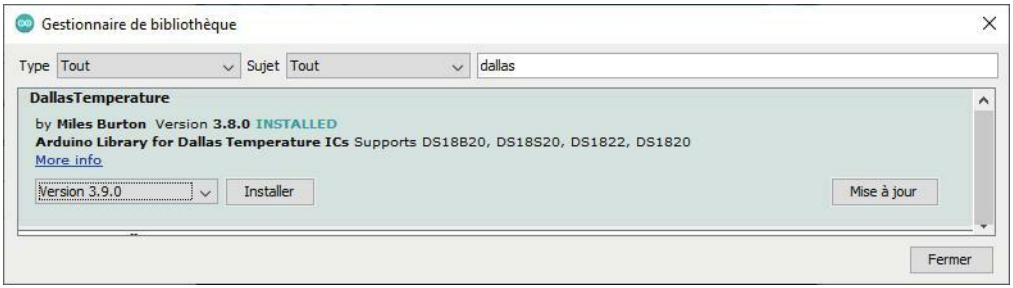
Elle indique à la fin le nombre de caractères dans la chaîne de caractères à envoyer. Nous verrons cela en détail lorsque nous examinerons le sketch Arduino. Avant cela, il nous faut installer une bibliothèque, car la thermistance *DS18B20* utilisée requiert un support particulier. Il s'agit d'un capteur bus One-Wire qui, comme son nom l'indique, envoie ses données via un

seul fil. Donc, en gros, deux bibliothèques sont nécessaires pour le sketch : la bibliothèque pour la thermistance DS18B20 et celle pour le bus One-Wire.

QU'EST-CE QU'UN BUS ONE-WIRE ?

1-Wire ou bus One-Wire est un bus à un conducteur avec une interface sérielle, conçu par la société Dallas Semiconductor Corp. Un fil de données sert à la fois pour l'alimentation en courant et comme fil d'émission et de réception.

Si vous installez la bibliothèque pour la DS18B20, il vous sera demandé lors de l'installation si vous souhaitez installer en même temps une autre bibliothèque. Vous devez répondre par oui. Dans le champ de recherche de la bibliothèque à ajouter, entrez alors *dallas* et cliquez sur le bouton *Installer*.



▲ **Figure 22-16**
Installation de la bibliothèque Dallas

OK, maintenant, on peut vraiment commencer !

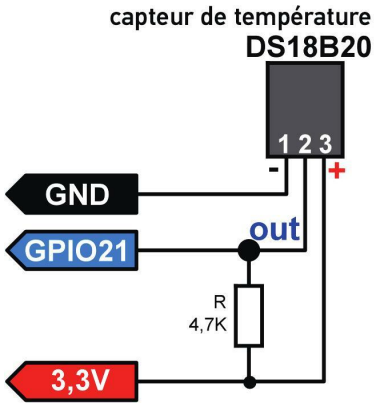
Schéma

Examinons d'abord la thermistance DS18B20. Capteur de grande précision, elle est capable de fonctionner avec une tension comprise entre 3 V et 5,5 V et couvre une plage de température allant de −55 °C à +125 °C. Ce capteur ressemble à un transistor à trois broches présentant la disposition suivante :

◀ **Figure 22-17**
Disposition des broches du DS18B20

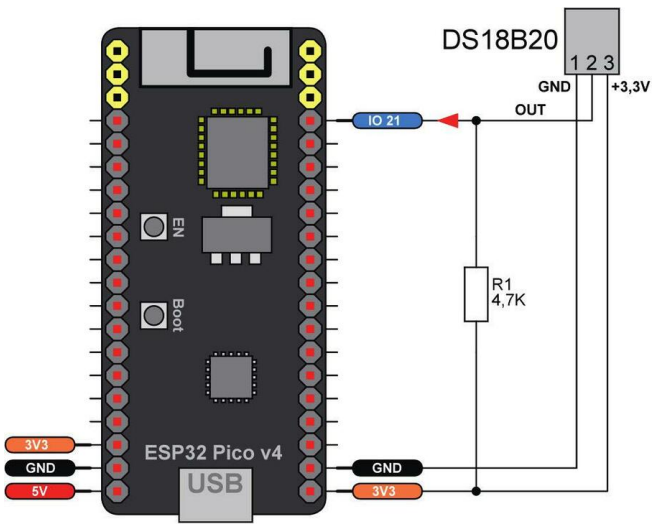
Puisqu'il s'agit d'un capteur communiquant via le *bus One-Wire*, vous avez la possibilité de raccorder plusieurs capteurs de ce genre si nécessaire. Nous commencerons toutefois avec un seul capteur qui, comme sur le bus I^2C, requiert une résistance de tirage (Pull-up) sur le bus, ce qui nous donne :

Figure 22-18 ▶
Circuit du DS18B20



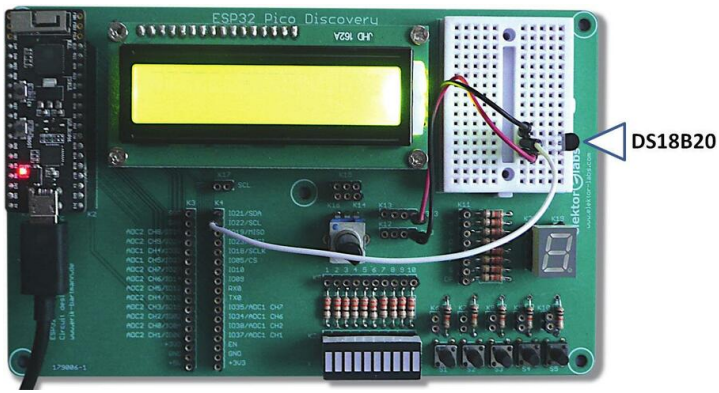
La résistance de 4,7 kΩ est la résistance de tirage (Pull-up) pour le bus. Il est possible ici de modifier le cas échéant la broche GPIO 21. Les lettres *GPIO* sont l'abréviation de *General Purpose Input/Output*, ce qui signifie qu'il s'agit d'une broche pouvant être programmée librement en tant qu'entrée ou sortie pour un usage général. Ne vous laissez pas impressionner, ce n'est rien d'autre qu'une broche de votre carte Arduino Uno, pouvant être configurée par l'instruction *pinMode* comme une entrée simple ou avec résistance de tirage (pull-up) ou une sortie. Ci-dessous le schéma des connexions avec l'ESP32-Pico-Board et la thermistance DS18B20 :

Figure 22-19 ▶
Schéma des connexions avec l'ESP32-Pico-Board et la thermistance DS18B20



Réalisation du circuit

Faire un circuit test sur l'ESP32-Pico-Discoveryboard est un jeu d'enfant. On y voit la thermistance DS18B20. La résistance de tirage (Pull-up) est cachée par un câble, mais elle est bien là !



◀ **Figure 22-20**
Circuit du DS18B20 sur l'ESP32-Pico-Discoveryboard

Sketch Arduino

Regardons maintenant de plus près le sketch Arduino. Je l'ai divisé en plusieurs parties pour que mes explications collent au plus près de ce qui se passe. La première partie va jusqu'à l'intégration supplémentaire de bibliothèques pour interroger la thermistance DS18B20. On a ajouté par ailleurs une bibliothèque qui supporte un client HTTP, car notre ESP32 est le client qui établit une liaison avec le serveur ThingSpeak. Remplacez ici les données d'accès et la clé API par vos données sinon la connexion ne pourra pas se faire :

```
#include <WiFi.h>              // Intégrer la bibliothèque Wi-Fi
#include <HTTPClient.h>        // Intégrer la bibliothèque client HTTP
#include <OneWire.h>           // Intégrer la bibliothèque One-Wire
#include <DallasTemperature.h> // Intégrer la bibliothèque Dallas
#define ONE_WIRE_BUS 21        // Broche DS18B20 – GPIO 21

const char* ssid         = "EriksWLan"; const char*
password                 = "Schnickschnack";
const char* serverName   = "http://api.thingspeak.com/update"; String
apiKey = "J7HKMQSSVU2QH6TK";
unsigned long lastTime = 0;
unsigned long timerDelay = 10000; // 10 secondes

OneWire oneWire(ONE_WIRE_BUS);      // Initialisation One-Wire
DallasTemperature sensors(&oneWire);// Transmettre référence One-Wire
                                    // à Dallas
```

```
void setup() { Serial.begin(115200);
  WiFi.begin(ssid, password);
  Serial.println("Connection au reseau Wi-Fi...");
  while(WiFi.status() != WL_CONNECTED) {
    delay(500);          // Courte pause
    Serial.print(".");  // Éditer point si aucune connexion
  }
  Serial.println("\nConnecté au réseau Wi-Fi "); Serial.print("IP
address: "); Serial.println(WiFi.localIP());
}
```

Tout le processus d'interrogation de la thermistance et de l'envoi de cette valeur de mesure a lieu au sein de la fonction `loop`. L'envoi des données dans le temps ne s'opère pas ici par la fonction `delay`, mais par la construction au moyen de la fonction `millis`, que vous connaissez déjà.

```
void loop() {
  // HTTP POST request toutes les 10 secondes
  if((millis() - lastTime) > timerDelay) {
    // Statut Wi-Fi OK ?
  if(WiFi.status()== WL_CONNECTED) {
    HTTPClient http;          // Client http
    http.begin(serverName);  // Serveur http
```

Voici quelques informations sur la méthode HTTP-Post que nous allons utiliser. Celle-ci envoie des données au serveur. Le type de corps, autrement dit le message à envoyer, dont le format est indiqué par l'en-tête `Content-Type`. Il existe différents types pour ce procédé. Le type *application/x-www-form-urlencoded* que nous utilisons a la signification suivante : les clés et les valeurs sont codées dans des paires clés-valeurs, séparées par *&*, avec un signe = entre la clé et la valeur. Vous vous souvenez certainement de cette partie de l'URL que j'ai envoyée au serveur ThingSpeak et qui se présente comme suit : *key=J7HKMNNSVU2- QH6TK&field1=4*. On y voit très bien les paires clés-valeurs. C'est justement cette configuration d'en-tête HTTP que nous obtenons avec la méthode `addHeader`.

La méthode `requestTemperatures` de la bibliothèque de capteurs sert à interroger la valeur de la température, tandis que la méthode `getTempCByIndex(0)` interroge le premier capteur avec index = 0 (même si nous n'en n'avons qu'un) et stocke la mesure dans les variables `temperature`. La demande HTTP à envoyer au serveur se compose alors de plusieurs informations comme la clé API, le champ de données et la valeur de mesure. Cette dernière est envoyée avec la méthode POST, une valeur de retour est ensuite attendue. Celle-ci fournit des renseignements sur le déroulement et le résultat de la demande. Un code retour de 200 indique que tout s'est bien passé et que les données ont été transmises au serveur. Toute autre valeur de retour indique une erreur, qu'il convient d'analyser.

```
      // Content-type header
      http.addHeader("Content-Type",
        "application/x-www-form-urlencoded");
      // Envoyer les données par HTTP-POST
      sensors.requestTemperatures(); // Demander mesure température
      float temperature = sensors.getTempCByIndex(0);
      String httpRequestData = "api_key=" + apiKey + "&field1=" +
        String(temperature);
      // HTTP-POST-Request
      int httpResponseCode = http.POST(httpRequestData);
      if(httpResponseCode == 200) {
        Serial.print("Température: ");
        Serial.println(temperature);
        Serial.println("SUCCESS - HTTP Response code: 200 - Données
                        envoyées !");
    }
    else {
       Serial.println("ERROR - Données non envoyées!");
       Serial.print("HTTP Response code: ");
       Serial.println(httpResponseCode);
      }
      // Débloquer les ressources
      http.end();
    }
    else {
      Serial.println("Wi-Fi Disconnected");
    }
    lastTime = millis();
  }
}
```

Veillez à régler correctement la vitesse de transmission au moniteur série à 115 200 bauds et observez ce qu'il affiche pendant la transmission des données. La figure ci-après montre un exemple de transfert de données.

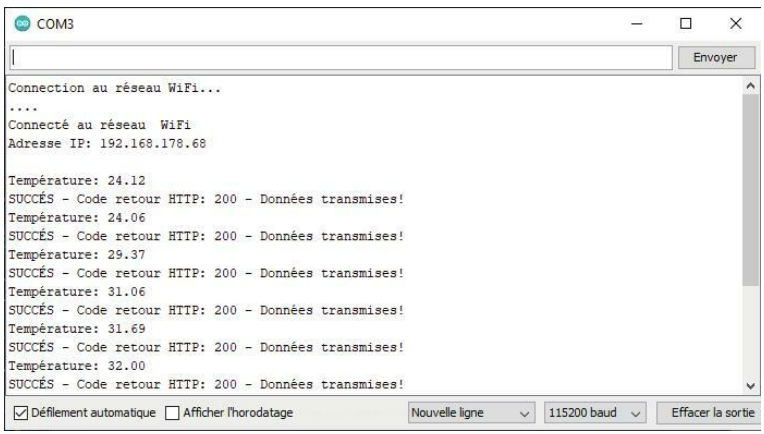


Figure 22-21
Le moniteur série fournit des renseignements sur le transfert de données en direction du serveur.

Vous pouvez suivre en même temps les données qui arrivent sur la page Internet ThingSpeak dans la chaîne correspondante. Le graphique suivant montre quelques valeurs que j'ai transmises en touchant la thermistance :

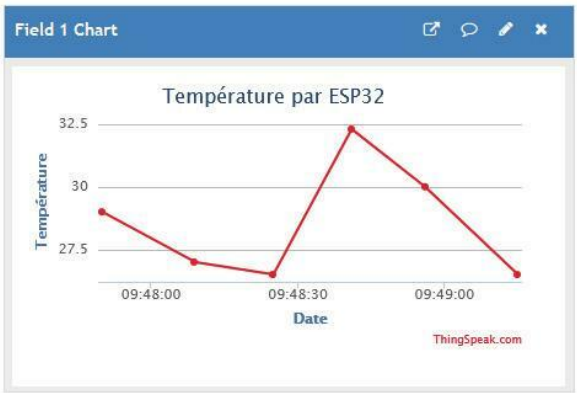


Figure 22-22 ▶
Valeurs dans la chaîne ThingSpeak

Veillez à ce que votre chaîne ne déborde pas de données lors des divers tests. Vous pouvez supprimer ces données pour repartir à zéro. Pour cela, allez dans l'onglet *Channel Settings* et déroulez le menu jusqu'en bas. Attention, vous devez cliquer sur le bon bouton rouge. Le bouton *Clear Channel* en haut permet de supprimer toutes les données antérieures, sans supprimer la chaîne.

Le bouton *Delete Channel* en bas supprime complètement la chaîne et avec elle, toutes les données qu'elle contient.

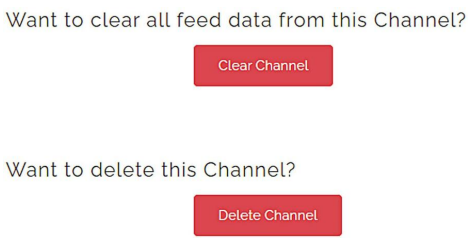


Figure 22-23 ▶
Vider ou supprimer la chaîne ThingSpeak

Problèmes courants

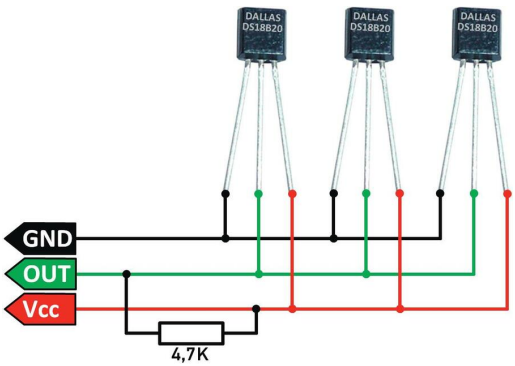
Si vous n'arrivez pas à établir la connexion entre votre carte ESP32 et le routeur, vérifiez encore une fois les données d'accès. Vous devez également saisir correctement la clé API pour que les valeurs de mesure puissent être transmises ensuite au serveur ThingSpeak. Vérifiez encore une fois ce que vous avez saisi. Si la thermistance DS18B20 ne livre pas de valeurs de mesure logiques, vérifiez une nouvelle fois le câblage ainsi que la broche GPIO utilisée sur l'ESP32-Pico-Board.

Qu'avez-vous appris ?

- Nous avons fait connaissance du composant électronique ou microcontrôleur ESP32 et appris comment le programmer avec l'IDE Arduino.
- Vous avez découvert deux cartes, sur lesquelles la puce ESP32 a été installée : l'ESP32-Pico-Board et la D1 R32.
- Vous avez raccordé une thermistance au microcontrôleur par le biais du bus One-Wire.
- Vous connaissez maintenant le service cloud ThinkSpeak et vous avez pu éditer une représentation graphique des données fournies en permanence par votre log de températures.

Workshop sur le log de températures

Il est possible de raccorder et d'interroger via le bus One-Wire plusieurs capteurs de température de type DS18B20. Il vous suffit de procéder comme suit :



◀ Figure 22-24
Plusieurs capteurs de température DS18B20 sur le bus One-Wire

Mais comment alors interroger un par un les différents capteurs ? Tous les modules ou esclaves connectés au bus I^2C se voient affectés leur propre adresse de bus. Cela nous permet de définir un adressage unique, ce qui ne semble pas être possible ici. Exact, mais une affectation précise est belle et bien effectuée. Chaque capteur dispose d'une adresse matérielle 64 bits unique, enregistrée dans une mémoire ROM interne. Voici un exemple de programmation possible par laquelle l'index est incrémenté à chaque échantillonnage de la température :

```
void loop() {
  ...
  float temperatur1;   // Enregistre température 1
  float temperatur2;   // Enregistre température 2
  ...

  // Demander mesure de la température
  sensors.requestTemperatures();
  temperatur1 = sensors.getTempCByIndex(0); // Capteur 1
  temperatur2 = sensors.getTempCByIndex(1); // Capteur 2
  ...
}
```

Modifiez votre sketch de façon que, dans ce cas, deux valeurs de mesure soient transmises au serveur ThingSpeak. Jusqu'à présent, vous n'avez envoyé qu'un seul champ via la requête HTTP. Celui-ci doit être modifié et étendu. Évidemment, vous devez faire aussi quelques ajustements sur la page du serveur et créer deux champs ou graphiques. Je n'en dirai pas plus. Cherchez de votre côté avant de me poser des questions. Vous apprendrez davantage ainsi que si l'on vous présente des solutions toutes faites. Il serait également intéressant de programmer un montage pour le capteur de température et d'humidité *DHT11* ou *DHT22* (voir montage n° 26) avec le module ESP32.

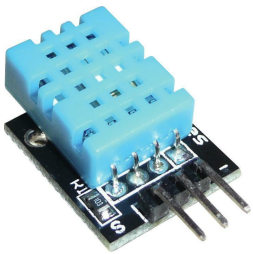


Figure 22-25 ▶
Le capteur de température et d'humidité DHT11 ou DHT22

Montage

23

Numérique appelle analogique

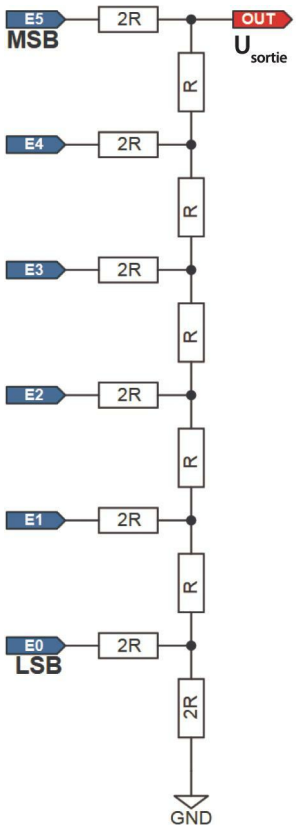
Dans ce montage, nous allons découvrir comment générer des signaux analogiques sur la carte Arduino Uno. Nous produirons diverses formes d'onde et nous fabriquerons un petit générateur de fonctions.

Comment convertir des signaux numériques en signaux analogiques ?

L'exploitation de signaux analogiques est relativement simple par les entrées analogiques avec votre carte Arduino. Le sens inverse – c'est-à-dire produire et distribuer une tension analogique à l'aide du microcontrôleur – n'est faisable qu'en utilisant les sorties numériques capables de MLI. Quand vous aurez vu la forme de la courbe des signaux MLI, vous saurez combien elle diffère de celle d'un signal analogique.

La plupart des microcontrôleurs courants ne convertissent pas un signal numérique en signal analogique. Il leur faudrait pour ce faire intégrer un convertisseur N/A. Notre projet consiste à fabriquer un tel convertisseur, appelé aussi CNA (*Digital-Analog-Converter* en anglais ou DAC), avec des moyens simples. Tout tourne ici autour du réseau R2R. Cette appellation est due au fait que le convertisseur est composé de plusieurs résistances, disposées en cascade et qui doivent se trouver entre elles dans un rapport déterminé. La disposition des éléments fait penser à une échelle, c'est pourquoi ce type de circuit est également nommé réseau en échelle de résistances dans la littérature spécialisée. Retenons seulement que le réseau de résistances sert à répartir une tension de référence qui, dans notre cas, est de +5 V. La figure 23-1 montre un réseau de résistances R2R avec une entrée de 6 bits.

Figure 23-1 ▶
Réseau de résistances R2R avec une entrée de 6 bits



Vous vous demandez peut-être d'où vient ce nom, R2R. Si vous regardez le schéma de plus près, vous verrez que les résistances n'ont pas une valeur fixée, et que seuls les rapports de résistance sont indiqués. Les résistances R2R (horizontales), qui sont reliées aux connexions E_0 à E_5 des sorties numériques, valent le double des résistances (verticales), qui relient les résistances précédentes et mènent au point de sortie U_{sortie}. La résistance du bas, qui est reliée à la masse, a la même résistance 2R que les résistances horizontales. La formule suivante peut être utilisée pour déterminer la tension de sortie :

$$U_{sortie} = \frac{U_{e5}}{2} + \frac{U_{e4}}{4} + \frac{U_{e3}}{8} + \frac{U_{e2}}{16} + \frac{U_{e1}}{32} + \frac{U_{e0}}{64}$$

Pour cet exemple avec six entrées, la résolution suivante peut être obtenue :

$$U_{résolution} = \frac{U_{ref}}{64}$$

U_{ref} est ici la tension avec laquelle les différentes entrées sont commandées. Pour une tension U_{ref} de 5 V, le résultat serait donc le suivant :

$$U_{résolution} = \frac{U_{ref}}{64} = \frac{5\ \text{V}}{64} = 78{,}13\ \text{mV}$$

Cette valeur représente le plus petit pas de progression obtenu chaque fois que la valeur binaire de l'entrée à 6 bits est incrémentée de 1. Le tableau 23-1 fournit les quatre premières valeurs ainsi que la dernière.

Valeur binaire	Tension de sortie
000000	0 V
000001	78,13 mV
000010	156,26 mV
000011	234,39 mV
...	...
111111	5 V

◀ Tableau 23-1
Combinaisons binaires et tensions de sortie arrondies

Nous avons donc, pour le shield de conversion N/A prévu, une définition de 6 bits ($2^6 = 64$). Vous avez peut-être remarqué que je n'ai donné jusqu'ici aucune valeur de résistance. Ce n'est pas utile tant que le rapport des résistances est exactement de 2:1. En outre, la tolérance des différentes résistances doit être aussi faible que possible pour obtenir des résultats relativement précis. Nous n'en tiendrons cependant pas compte dans ce montage.

Composants nécessaires

Ce montage nécessite les composants suivants.

Composant	
17 résistances de 47K	
1 jeu de connecteurs femelles empilables • 2 connecteurs à 8 broches • 1 connecteur à 6 broches • 1 connecteur à 10 broches	
1 carte shield	

◀ Tableau 23-2
Liste des composants

Le réseau R2R avec rapports de résistance 2:1 peut s'avérer difficile à réaliser, car vous devez trouver des valeurs de résistance qui sont dans ce rapport entre elles. La solution n'est certes pas simple. J'ai choisi une résistance de 47K pour que les courants en circulation ne soient pas trop élevés. Vous vous demandez peut-être si une résistance de 23,5K existe. Non seulement je ne crois pas, mais cette valeur est en plus très facile à obtenir. Quand on branche deux résistances de même valeur en parallèle, le résultat obtenu est l'exacte moitié de la résistance en question. Donc si $R_1 = R_2$, on obtient :

$$\frac{1}{R_{totale}} = \frac{1}{R_1} + \frac{1}{R_2} = \frac{1}{R} + \frac{1}{R} = \frac{2}{R}$$

$$\text{donc } R_{totale} = \frac{R}{2}$$

C'est élémentaire, n'est-ce pas ? Examinons maintenant le schéma.

Schéma

Vous constaterez que pour des questions de place, les résistances ont été disposées à l'horizontale.

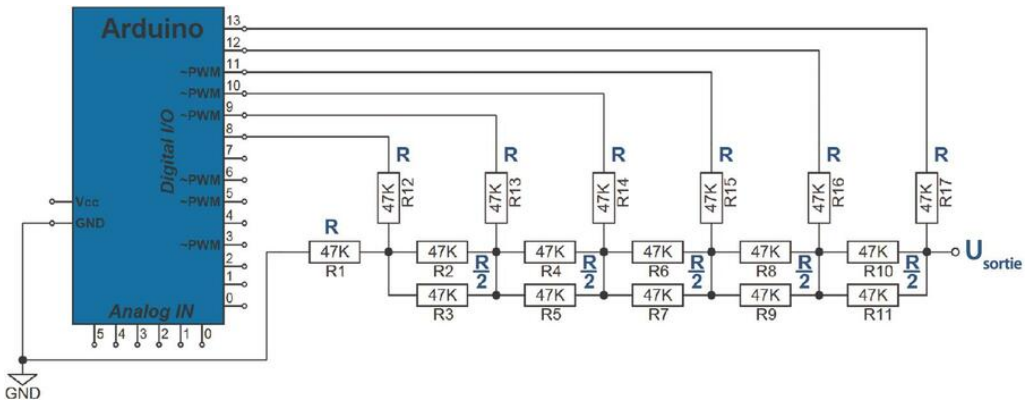
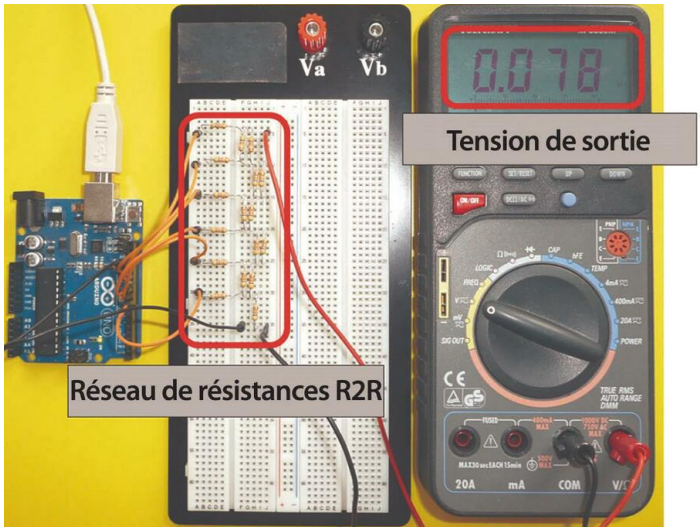


Figure 23-2 ▲
Schéma du générateur de signaux analogiques

Les résistances R ont naturellement une valeur de 47K. Les paires de résistances R/2 ont comme valeur résultante 23,5K.

Réalisation du circuit

Le circuit peut évidemment aussi être réalisé sans shield, sur une plaque d'essais.



◀ **Figure 23-3**
Réseau R2R sur une plaque d'essais

Maintenant que nous avons élucidé les questions techniques, nous allons pouvoir nous pencher sur le sketch.

Sketch Arduino

```
int pinArray[] = {8, 9, 10, 11, 12, 13};
byte R2RPattern;
void setup() {
  for(int i = 0; i < 6; i++)
    pinMode(pinArray[i], OUTPUT);
  R2RPattern = B000001; // Configuration binaire pour commander les sor-
ties numériques
}

void loop() {
  for(int i = 0; i < 6; i++){
    digitalWrite(pinArray[i], bitRead(R2RPattern, i) == 1?HIGH:LOW);
  }
}
```

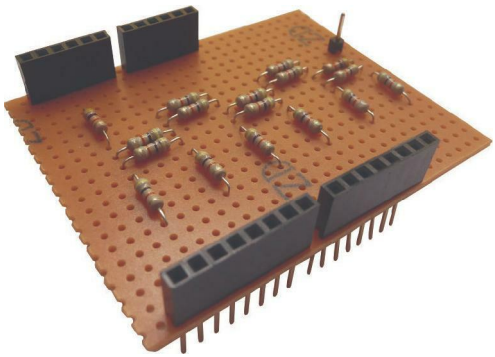
Ce sketch, vraiment très court, commande les sorties numériques sur lesquelles se trouve le réseau R2R. Elles sont commandées via la variable `R2RPattern`, qui délivre une tension correspondante à la sortie du réseau.

Revue de code

Le réseau R2R est commandé avec la combinaison de bits `B000001` tirée du sketch ; le multimètre affiche une tension de 0,078 V, soit 78 mV. Dans le tableau 23-1, la valeur est de 78,13 mV pour la combinaison de bits. La valeur de sortie de 78 mV ne correspond donc pas tout à fait à la valeur calculée du tableau, mais c'est tout de même correct quand on sait que le résultat se trouve par exemple légèrement faussé par les tolérances matérielles des résistances utilisées ou par des erreurs d'affichage du multimètre. Il m'est arrivé de construire un réseau R2R où les valeurs coïncidaient presque toutes jusqu'à deux chiffres après la virgule, mais ce n'était que pur hasard. L'interrogation des bits du nombre du pattern est effectuée au moyen de la fonction `bitRead`. Pour placer les bits au moyen de la fonction `digitalWrite` dans la boucle `for`, nous utilisons un *opérateur ternaire* représenté par un point d'interrogation. Nous l'avons déjà évoqué dans le montage n° 10 du dé électronique. Si la condition formulée `bitRead(R2RPattern, i) == 1` renvoit la valeur `true`, la valeur qui se trouve immédiatement après le point d'interrogation (`HIGH`) est renvoyée à son tour. Si tel n'est pas le cas, c'est la valeur qui figure après les deux points (`LOW`) qui est renvoyée.

Réalisation du shield

Figure 23-4 ▶
Réalisation du réseau R2R sur un shield dédié



Cette figure montre bien le réseau de résistances, la broche en haut du shield étant la sortie sur laquelle vous pouvez brancher votre multimètre pour mesurer la tension de sortie.

Commande du registre de port

Je ne vous apprendrai rien en vous disant que la carte Arduino ne communique que par les entrées et les sorties. Ceci vaut également pour commander des LED, des moteurs, des servomoteurs et pour lire, entre autres, les valeurs d'un capteur de température ou d'une résistance réglable ou photosensible. Votre microcontrôleur ATmega328P travaille en interne avec ce qu'on appelle des *registres*, raccordés aux entrées et sorties (broches). Dans le domaine informatique, ce sont des zones de mémoire à l'intérieur d'un processeur, qui sont reliées directement à l'unité centrale de calcul. Ainsi l'accès à ces zones est très rapide puisque le détour par des circuits mémoire externes est évité. Les différentes broches de votre carte Arduino sont reliées en interne à des registres de port, encartouchés en couleurs (vert, rouge ou jaune) et nommés port B, C ou D sur la figure 23-5.

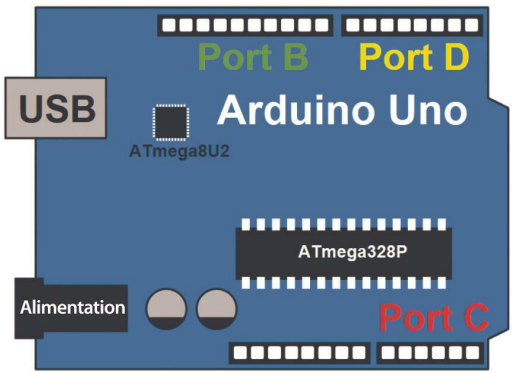


Figure 23-5
Registres de port de la carte Arduino

Regardons par exemple le port B de plus près sur la figure 23-6.

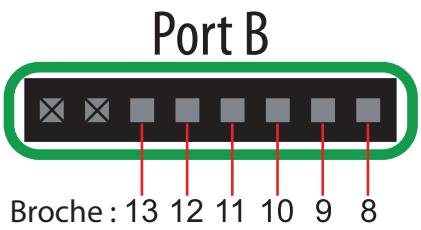


Figure 23-6
Registre de port B

On reconnaît immédiatement les entrées et sorties numériques (broches 9 à 13). Les deux broches de gauche sont pour nous sans intérêt, car elles sont reliées à *Aref* (entrée pour la tension de référence du convertisseur analogique-numérique) et à la masse et ne peuvent être manipulées. Six bits en tout sont donc disponibles dans le registre de port B. Ils vont nous

servir à faire des choses diverses. Comme par hasard, notre réseau de résistances est lui aussi commandé avec 6 bits. Je ne vous en dis pas plus pour l'instant. Chacun des trois ports est sollicité dans un sketch au moyen des identifiants suivants :

- `PORTB`
- `PORTC`
- `PORTD`

Nous savons déjà comment les différents ports sont sollicités, mais nous nous en tiendrons, comme je l'ai dit plus haut, au port B dans notre exemple.

Nous avons vu que quand on programme des broches numériques, on doit définir dans la fonction `setup` si elles vont servir d'entrée ou de sortie. Mais dans le cas d'un registre de port, comment lui dire de servir d'entrée ou de sortie ? Eh bien, je vais vous l'expliquer mais je dois vous dire quelque chose auparavant : vous pouvez bien sûr attribuer à chaque bit du registre de port un sens de circulation des données particulier. Le registre complet ne fonctionne pas de telle sorte que toutes les broches servent d'entrées ou de sorties. Chaque broche peut être configurée séparément. D'autres registres sont en effet spécialement prévus pour influer sur le sens de circulation des données de chaque broche. Ils ont pour nom DDRx, x indiquant le port à solliciter. Le registre DDRB est par conséquent celui de notre `PORTB` – (DDR, pour *Data Direction Register*, signifie à peu de chose près registre de direction de données). Voyons comment tout cela fonctionne dans le détail. Avant d'utiliser un port, je dois donc définir le sens de circulation des données par le DDR correspondant. Sur la figure 23-7, les flèches indiquent les sens de circulation des données que nous voulons obtenir avec notre programmation.

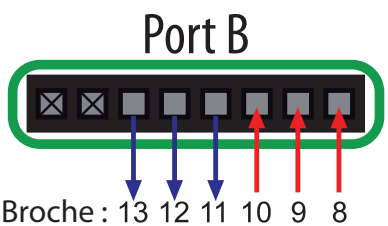


Figure 23-7 ▶ Registre de port B avec divers sens de circulation des données selon les broches

La configuration est donc la suivante :

- entrées : broches 8, 9 et 10 ;
- sorties : broches 11, 12 et 13.

Pour affecter un sens de circulation des données à une broche, la valeur du tableau suivant doit être entrée dans le DDR.

Valeur	Mode de fonctionnement
0	Broche servant d'entrée, comparable à `pinMode(pin, INPUT);`
1	Broche servant de sortie, comparable à `pinMode(pin, OUTPUT);`

Tableau 23-3
Valeurs pour le registre DDR

Cela nous donne pour le DDR la programmation de la figure 23-8.

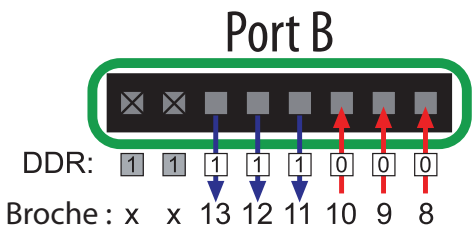


Figure 23-8
Initialisation du DDR pour les différents sens de circulation des données

Nous pouvons maintenant mettre par exemple les sorties numériques des broches 11, 12 et 13 au niveau `HIGH` par l'instruction `PORTB`. Voici un extrait correspondant tiré d'un sketch :

```
void setup() {
  DDRB =  0b11111000; // Broches 8, 9, 10 = INPUT.
                      // Broches 11, 12, 13 = OUTPUT
  PORTB = 0b00111000; // Mise des broches 11, 12, 13 au niveau HIGH
}

void loop(){ /* vide */ }
```

Les deux bits les plus significatifs pour les broches non utilisées ont simplement été pourvus d'un 1 dans le DDR. Cela n'a aucune importance dans notre cas. Si vous regardez la mise des sorties au niveau `HIGH`, que constatez-vous de différent par rapport à la manipulation des broches habituelle ? Je vous donne les deux variantes pour comparaison :

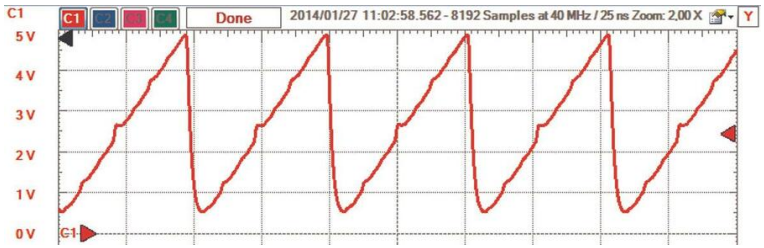
```
digitalWrite(11, HIGH);
digitalWrite(12, HIGH);
digitalWrite(13, HIGH);
PORTB = 0b00111000;
```

Aucune idée ? Bon. La manière traditionnelle de gauche met les différentes broches l'une après l'autre au niveau `HIGH`. Celle de droite met par contre toutes les broches en même temps au niveau `HIGH` avec une seule instruction, la configuration binaire étant alors appliquée simultanément à toutes les broches. Mieux vaut donc choisir la nouvelle variante de manipulation

des ports pour aller plus vite. Le sketch suivant génère une tension en dents de scie à la sortie du réseau de résistances :

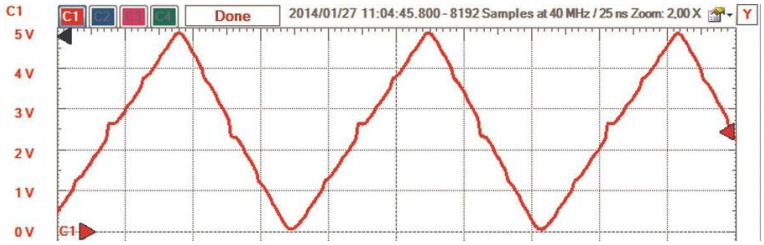
```
void setup() {
 DDRB =  0b11111111; // Toutes les broches programmées en tant que sorties
}

void loop() {
  for(int i = 0; i <= 63; i++) // 63 = B00111111
    PORTB = i; // Commande du registre de port B
}
```

Figure 23-9 ▶ Oscillogramme avec une courbe en dents de scie



D'après vous, comment adapter le sketch pour obtenir la courbe suivante ?

Figure 23-10 ▶ Oscillogramme avec une courbe en triangles



Bien joué si vous avez trouvé la solution !

```
void loop() {
  for(int i = 0; i <= 63; i++)
    PORTB = i; // Commande du registre de port B //(front montant)
  for(int i = 63; i >= 0; i--)
    PORTB = i; // Commande du registre de port B (front descendant)
}
```

Quelles sont les autres courbes ? Qu'en est-il d'une courbe sinusoïdale ? La fonction sinus ayant besoin d'un certain temps pour calculer les valeurs, on a eu l'idée de créer des tables de correspondance ou *Lookup-Tables* (LUT). Les résultats d'un calcul y sont déjà enregistrés. On peut ainsi reproduire la courbe d'une fonction sinus en s'aidant des points qui se trouvent sur la courbe, par exemple.

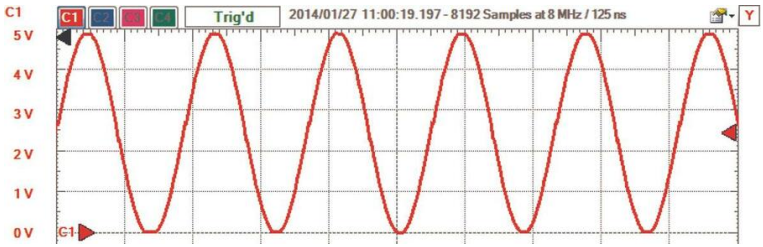


Figure 23-11 Oscillogramme avec une courbe sinusoïdale

Le sketch pour générer la courbe sinusoïdale est plutôt laborieux à écrire car la LUT est très longue.

```
byte LUT[] =
{31, 32, 32, 33, 33, 34, 34, 35, 35, 36, 36, 37, 38, 38, 39, 39, 40, 40,
 41, 41, 42, 42, 43, 43, 44, 44, 45, 45, 46, 46, 47, 47, 48, 48, 49, 49,
 50, 50, 50, 51, 51, 52, 52, 52, 53, 53, 54, 54, 54, 55, 55, 55, 56, 56,
 56, 57, 57, 57, 58, 58, 58, 59, 59, 59, 59, 60, 60, 60, 60, 60, 61, 61,
 61, 61, 61, 61, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62,
 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 62, 61, 61, 61,
 61, 61, 61, 60, 60, 60, 60, 60, 59, 59, 59, 59, 58, 58, 58, 57, 57, 57,
 56, 56, 56, 55, 55, 55, 54, 54, 54, 53, 53, 52, 52, 52, 51, 51, 50, 50,
 50, 49, 49, 48, 48, 47, 47, 46, 46, 45, 45, 44, 44, 43, 43, 42, 42, 41,
 41, 40, 40, 39, 39, 38, 38, 37, 36, 36, 35, 35, 34, 34, 33, 33, 32, 32,
 31, 30, 30, 29, 29, 28, 28, 27, 27, 26, 26, 25, 24, 24, 23, 23, 22, 22,
 21, 21, 20, 20, 19, 19, 18, 18, 17, 17, 16, 16, 15, 15, 14, 14, 13, 13,
 12, 12, 12, 11, 11, 10, 10, 10, 9, 9, 8, 8, 8, 7, 7, 7, 6, 6, 6, 5, 5,
 5, 4, 4, 4, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6,
 7, 7, 7, 8, 8, 8, 9, 9, 10, 10, 10, 11, 11, 12, 12, 12, 13, 13, 14, 14,
 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23,
 24, 24, 25, 26, 26, 27, 27, 28, 28, 29, 29, 30, 30, 31};

void setup() {
  DDRB = 0b11111111; // Toutes les broches programmées en tant que sorties
}

void loop() {
  for(int i = 0; i <= 360; i++)
    PORTB = LUT[i]; // Commande du registre de port B
}
```

VÉRIFIEZ BIEN VOTRE SKETCH !

En effet, vous courez le risque non négligeable de programmer votre microcontrôleur de telle sorte qu'il ne réagisse plus par la suite. Si vous regardez le port D, vous verrez que les signaux de contrôle RX et TX se trouvent respectivement sur les broches 0 et 1. RX sert à recevoir et TX à envoyer les données. Le sens de circulation des données est donc le suivant : RX = `INPUT` et TX = `OUTPUT`. Si vous modifiez par inadvertance la programmation de ces valeurs via DDRD, vous ne pourrez à coup sûr plus transmettre aucun sketch sur votre carte Arduino. Vous devez par conséquent être sûr de ce que vous faites. Mieux vaut vérifier trois fois votre sketch avant de l'envoyer au microcontrôleur. Vous trouverez des informations plus précises sur le lien :

https://www.arduino.cc/en/Reference/PortManipulation

Problèmes courants

Si la tension de sortie du réseau R2R ne correspond pas aux valeurs souhaitées pour la combinaison binaire, vérifiez les points suivants :

- Toutes les résistances utilisées pour le réseau R2R ont-elles la même valeur ?
- Aucune connexion au réseau n'a été oubliée ? (Je sais de quoi je parle, car j'avais oublié un point de jonction, et j'ai passé près de dix minutes à chercher l'erreur !)

Exercice complémentaire

Essayez, en ajustant le tableau LUT, de créer des courbes de formes différentes. Vérifiez dans tous les cas que seuls 6 bits sont à votre disposition pour représenter une courbe. Le domaine de valeurs va de 0 à 63. Si vous êtes au-dessus, ni le circuit ni votre microcontrôleur n'en souffriront, mais la courbe ne ressemblera à coup sûr pas à ce que vous auriez souhaité. Une autre méthode également très intéressante consiste à utiliser le circuit d'extension de port MCP23017 avec ses 16 broches d'E/S. Cela permet de profiter d'une résolution bien plus élevée des différentes valeurs de tension

par bit. De plus, pour une meilleure stabilité de sortie du réseau de résistances R2R, vous pouvez aussi utiliser l'amplificateur opérationnel LM358.

Qu'avez-vous appris ?

- Ce montage vous a présenté un réseau de résistances R2R.
- Avec ce réseau, vous avez pu réaliser un convertisseur numérique/analogique simple.
- Vous avez découvert les registres de port de votre carte Arduino et manipulé les sorties numériques au moyen du port B.

Montage

24

Programmer Arduino avec un langage de programmation par blocs

Il existe d'autres alternatives non moins intéressantes que la programmation de la carte Arduino avec l'environnement de développement Arduino. Peut-être avez-vous entendu parler de l'environnement de programmation *Scratch*. Ce langage de programmation permet de créer très facilement un programme ou un script sans avoir à entrer une seule ligne de code. Nous allons voir plus précisément comment cela est possible. Quel sera mon objectif dans ce montage ? Eh bien, tout d'abord vous initiez au langage Scratch ou *S4A*, qui est l'abréviation de *Scratch for Arduino*, et vous permettre ensuite de passer à d'autres *langages de programmation par blocs*. Outre S4A, nous aborderons ArduBlock et Open Roberta Lab, qui fonctionnent de manière similaire. J'aurai également l'occasion de vous présenter la puissante infrastructure IoT Node-RED, qui est aussi une programmation avec une assistance visuelle.

S4A – Scratch for Arduino

S4A est une extension de Scratch, spécialement développée pour programmer la carte Arduino. Dans S4A, toutes les opérations disponibles sont organisées en catégories ou palettes réparties selon des couleurs :

◀ **Figure 24-1**
Les catégories d'instruction dans S4A

Comme dans tous les autres langages par blocs qui offrent une assistance visuelle, les instructions disponibles sont proposées avec ce genre de structure et il n'est pas nécessaire de les apprendre par cœur, car il est toujours possible de voir leur fonction. Cette manière de procéder permet de programmer ou de structurer le projet de manière intuitive, ce qui devient un jeu d'enfant avec le temps. Grâce à l'organisation par couleurs, le cerveau se fait une image des algorithmes, ce qui facilite énormément la compréhension de la programmation non seulement pour les débutants, mais aussi pour les personnes plus expérimentées. S4A étant une extension de Scratch, dont la priorité n'était pas à l'origine de rendre plus simple la programmation de la carte de prototypage Arduino mais de seulement obtenir des résultats visuels sur un écran, les différentes catégories et leur désignation peuvent parfois dérouter. La catégorie *mouvement* peut être comprise au sens figuré comme une catégorie d'action pour la carte Arduino, qui déplace quelque chose sur la carte Arduino. Elle contient par exemple toutes les actions Arduino visant à interroger des capteurs ou à agir sur des broches analogiques ou numériques.

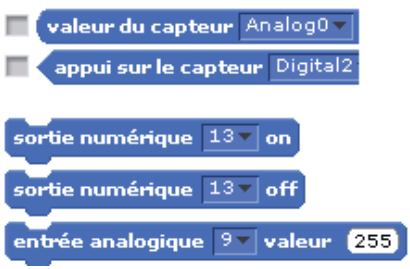


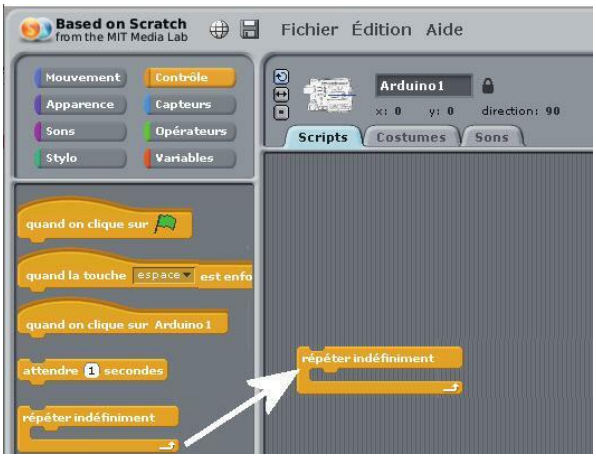
Figure 24-2 ▶
Les catégories d'instruction dans S4A – extrait

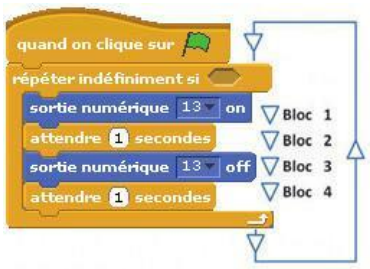
Dans S4A, les instructions qui doivent être exécutées sont représentées par des blocs fonctionnels, appelés simplement blocs, que nous pouvons assembler comme dans un puzzle. Ces blocs sont prélevés d'une réserve prédéfinie (de catégories ou palettes) et assemblés dans des séquences logiques. Vous obtenez alors un organigramme facile à comprendre grâce à ces composants visuels. Le positionnement ou l'assemblage des différents blocs se fait par glisser-déposer, c'est-à-dire en faisant glisser le bloc avec le bouton gauche de la souris et en le relâchant à la position souhaitée. Les blocs ne s'imbriquent que si la logique de programmation est correcte. Vous voyez ensuite la confirmation de l'imbrication des blocs, vous indiquant que vous êtes sur la bonne voie.

Sur la figure 24-3 du haut, je fais glisser le bloc Répéter indéfiniment pris dans la palette Contrôle vers la partie droite nommée Scripts.

Sur la figure 24-3 du bas, vous voyez quelques-uns de ces blocs, qui présentent un creux et un nez permettant de les relier au bloc voisin. S4A permet de réaliser une boucle sans fin qui met alternativement la bro-

che numérique 13 au niveau HIGH ou LOW. Une pause d'une seconde est observée entre ces actions. Le script que je vous montre ici est bien sûr très simple et fait office d'introduction. Il s'agit d'un *script Hello World*.





◀ **Figure 24-3**
Placement et assemblage de blocs

Le script démarre lorsque vous cliquez sur le drapeau vert qui se trouve tout en haut, comme pour donner le départ. Tous les blocs portent une inscription qui identifie clairement leur fonction pour faciliter la programmation aux débutants. Tous les langages de programmation par blocs avec une aide visuelle s'orientent sur ce schéma et même s'il y a des différences dans l'apparence ou la fonctionnalité des différents langages, on s'y retrouve très vite. Je dirais presque que si on en connaît un, on les connaît tous ! J'aimerais donc commencer ce montage avec S4A. Vous trouverez le site Internet de S4A à l'adresse :

http://s4a.cat/

Le principe de S4A

Avant de faire vos premiers pas, permettez-moi de vous expliquer la communication entre S4A et votre carte Arduino. Comment fonctionne l'ensemble et selon quel principe ? Pour répondre à ces questions, je vais

tout d'abord vous rappeler le principe de la programmation Arduino avec l'environnement de développement Arduino.

Étape 1 : développer un sketch et le transférer sur la carte Arduino

La première phase consiste à programmer pour une tâche un sketch qui sera ensuite transféré sur la carte Arduino.

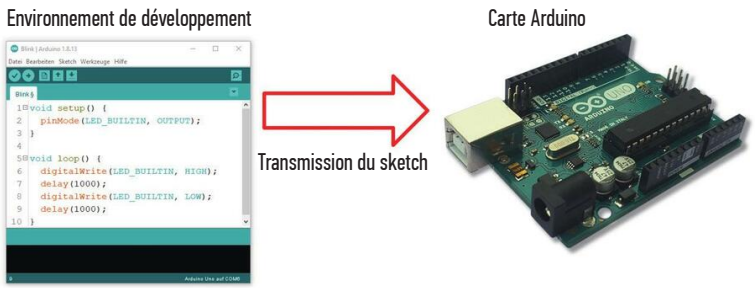


Figure 24-4 ▶
Le transfert du sketch sur la carte Arduino

Une fois le téléchargement réussi, l'environnement de développement n'a normalement plus rien à faire et la carte Arduino peut être débranchée de l'ordinateur en présence d'alimentation électrique, à moins que vous vouliez surveiller l'interface sérielle avec le moniteur série et recevoir ou envoyer des données. Pour la recherche d'erreur, cela s'avère très utile.

Étape 2 : la carte Arduino peut fonctionner de manière autonome avec son sketch téléchargé

Si maintenant vous débranchez votre carte Arduino du port USB, elle ne sera évidemment plus sous tension et stoppera le travail.

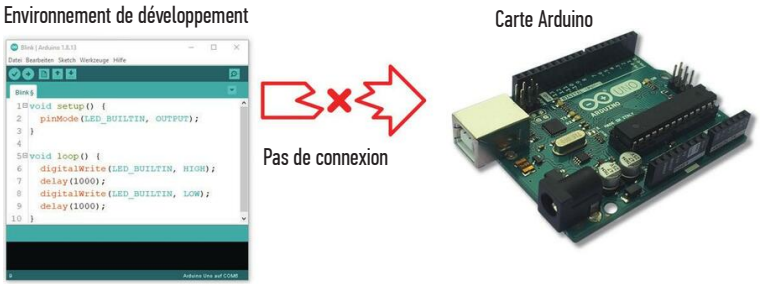
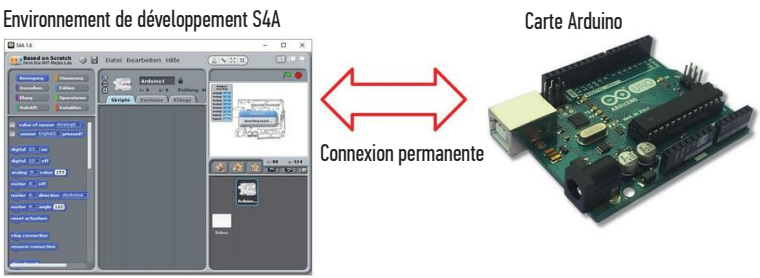


Figure 24-5 ▶
L'environnement de développement n'a plus rien à faire après le téléchargement du sketch.

La prise d'alimentation électrique vous permet de garder la carte Arduino sous tension, ce qui fait que celle-ci fonctionne de manière absolument autarcique et indépendamment de l'ordinateur avec lequel elle a été programmée. Il en va autrement pour la programmation avec S4A.

L'environnement de développement S4A fonctionne différemment de l'environnement de développement Arduino. Il doit maintenir un contact permanent avec la carte Arduino. Avec la méthode classique de l'environnement de développement Arduino, la carte est programmée avec un sketch spécifique pour une tâche spécifique. Le contrôle incombe donc à la carte Arduino elle-même. Avec S4A, nous avons chargé pour la communication un firmware qui doit rester disponible en permanence, indépendamment de la tâche en cours. Il y a donc une liaison permanente entre l'environnement de développement S4A (Scratch) et le firmware installé. Le contrôle n'incombe plus à la carte mais à S4A.



◀ Figure 24-6
L'environnement de développement S4A doit garder en permanence le contact avec la carte Arduino.

Vous pouvez télécharger ici le firmware nécessaire à la version 1.6 de S4A :

http://s4a.cat/downloads/S4AFirmware16.ino

Le firmware pour la carte Arduino est :

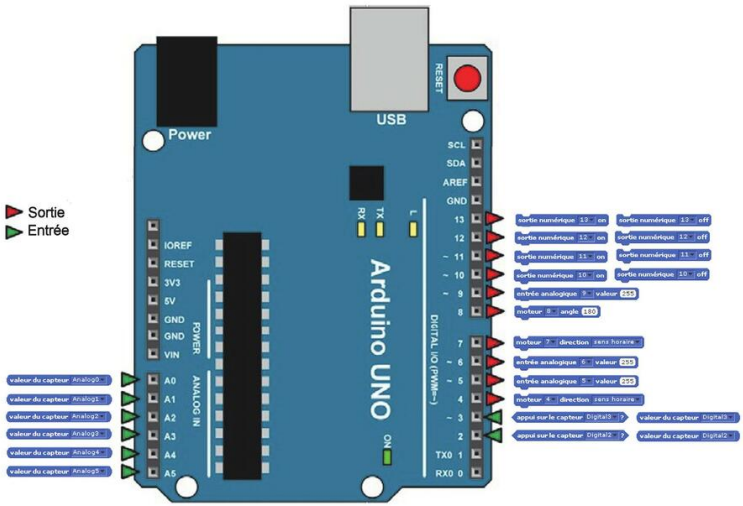
```
S4AFirmware16.ino
```

Vous devez lancer ce programme avec votre environnement de développement Arduino. Lorsque vous avez transféré ce firmware comme sketch sur votre carte Arduino et que vous lancez ensuite l'application S4A, celle-ci doit d'abord déterminer si une carte Arduino préparée avec le firmware requis est accessible. Vous pouvez suivre cette procédure dans le coin supérieur droit de S4A. Vous y verrez la boîte de dialogue représentée sur la figure suivante :

◀ Figure 24-7
L'environnement de développement S4A cherche une carte Arduino avec le firmware approprié.

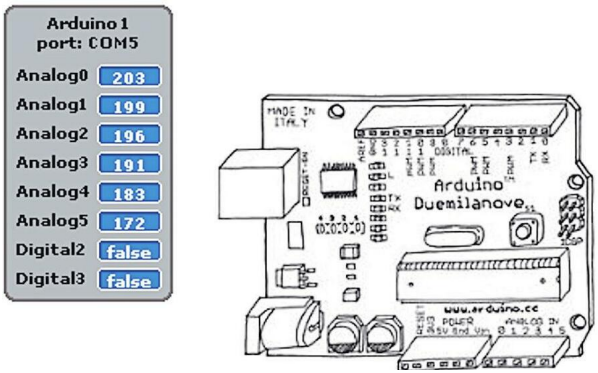
Une fois la carte détectée, cette boîte de dialogue disparaît et les entrées analogiques fournissent des valeurs aléatoires qui s'affichent. Cela montre que la communication entre S4A et votre carte Arduino fonctionne. J'ai représenté sur la figure ci-après les principaux blocs à côté de la carte Arduino et ses broches. Le site Internet S4A fournit des détails supplémentaires.

Figure 24-8 ▶
Les connexions de la carte Arduino et les blocs S4A correspondants



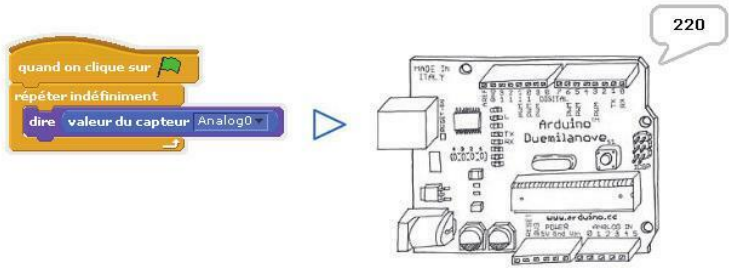
Si vous souhaitez interroger par exemple une entrée analogique, vous devez alors employer un des blocs *valeur du capteur*. Pour l'entrée analogique *A0,* celui-ci se présente comme indiqué ci-après. À l'aide du petit triangle noir à côté de la désignation *Analog0*, vous pouvez ouvrir une liste contenant tous les canaux disponibles.

Pour obtenir une vue d'ensemble complète de toutes les entrées, c'est-à-dire des entrées analogiques A0 à A5 et des entrées numériques D2 à D3, il vous suffit de jeter un œil dans S4A. Cela fonctionne sans que vous ayez besoin de démarrer un script. Par exemple :



◀ Figure 24-9
Vue d'ensemble des entrées de la carte Arduino

Un script Scratch très simple pour afficher une entrée analogique peut se présenter comme suit :



◀ Figure 24-10
Affichage d'une valeur analogique

Cliquez sur le drapeau vert pour afficher dans une bulle la valeur analogique du canal A0.

Script à télécharger

Que diriez-vous d'un script qui vous fournirait l'affichage suivant et qui vous présenterait par exemple quatre entrées analogiques sous la forme de courbes temporelles ? Ce script est disponible à l'adresse :

https://www.editions-eyrolles.com/dl/0100583

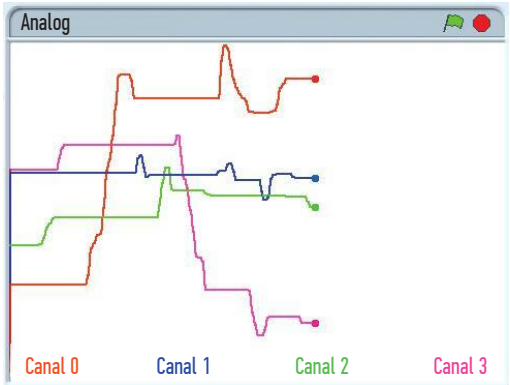


Figure 24-11 ▶
Le suivi analogique avec quatre canaux

ArduBlock – Arduino par blocs

J'aimerais vous présenter dans ce montage un autre langage de programmation par blocs avec une aide visuelle : *ArduBlock*. La différence avec S4A est qu'il n'y a pas besoin d'avoir une connexion permanente entre l'ordinateur et Arduino, car un sketch Arduino correspondant est généré (en fonction du script ArduBlock créé), qui est ensuite téléchargé individuellement sur la carte Arduino. La version 2 d'ArduBlock est disponible depuis peu et offre, une fois installée correctement, une intégration dans l'environnement de développement Arduino. Le fichier d'exécution d'ArduBlock est un fichier d'archives Java avec l'extension *.jar*. Les fichiers JAR sont principalement utilisés pour distribuer des bibliothèques de classes et des programmes Java. Ils sont comparables à des fichiers ZIP. Ce fichier peut être téléchargé à l'adresse Internet :

https://github.com/letsgoING/ArduBlock2

Cliquez sur le bouton vert *Code* puis sur *Download ZIP* :

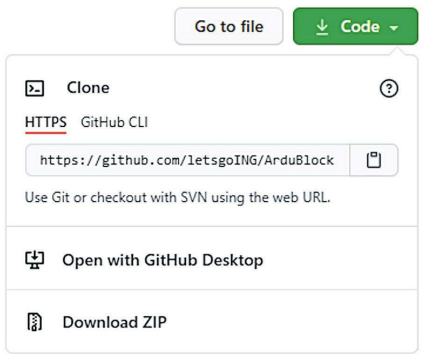


Figure 24-12 ▶
Le téléchargement du fichier ArduBlock

Procédez comme suit pour exécuter l'intégration dans l'environnement de développement Arduino :

Étape 1 : décompresser le fichier ZIP téléchargé

Le fichier ZIP doit être décompressé par un programme usuel comme 7ZIP de façon à obtenir la structure de dossiers suivante dans le répertoire correspondant (j'ai entouré en rouge les informations qui nous importent) :

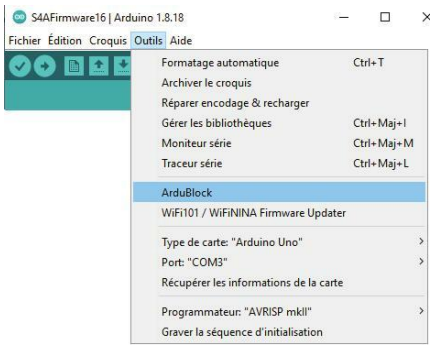
Étape 2 : copier la structure de dossiers dans le répertoire d'outils Arduino

Pour que l'environnement de développement Arduino prenne connaissance d'ArduBlock, vous devez copier la structure de dossiers marquée (fichier JAR inclus) *ArduBlock\tool\ardublock_letsgoing_21.jar* dans le répertoire *tools* de l'installation Arduino. Vous obtenez :

Redémarrez impérativement l'IDE Arduino pour intégrer la prise en charge ArduBlock.

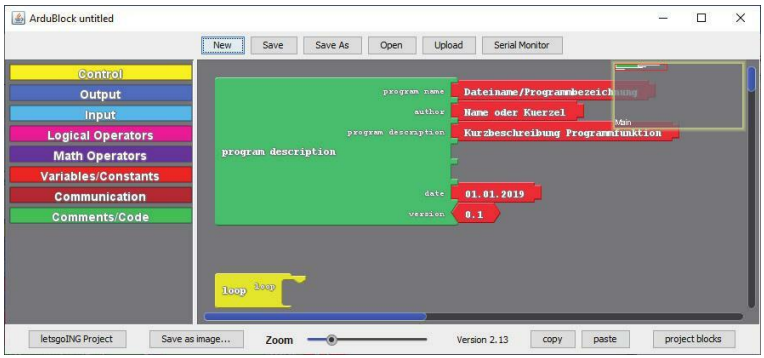
Étape 3 : démarrer l'environnement de développement Arduino et sélectionner ArduBlock

Ouvrez l'application ArduBlock dans l'environnement de développement Arduino via l'option du menu *Outils* :



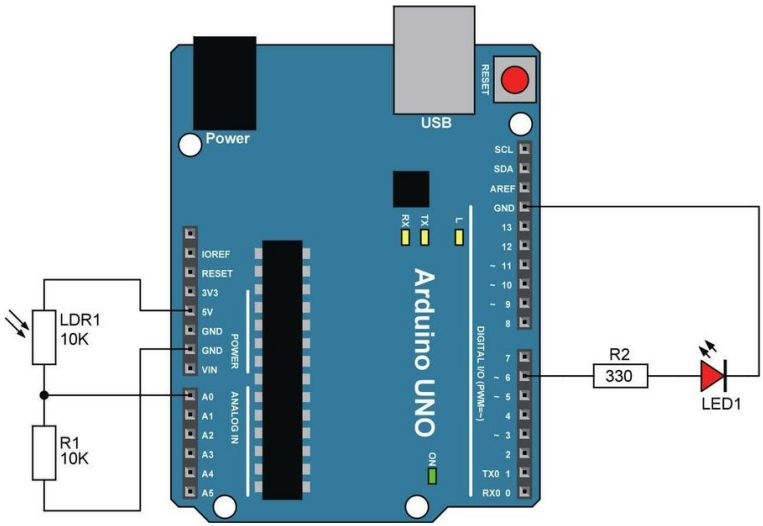
Lors de l'utilisation d'ArduBlock, veillez à sélectionner le bon port COM. Tout est maintenant installé et vous pouvez commencer à programmer avec ArduBlock. L'environnement de développement prend l'apparence suivante :

Figure 24-13 ▶
L'application ArduBlock

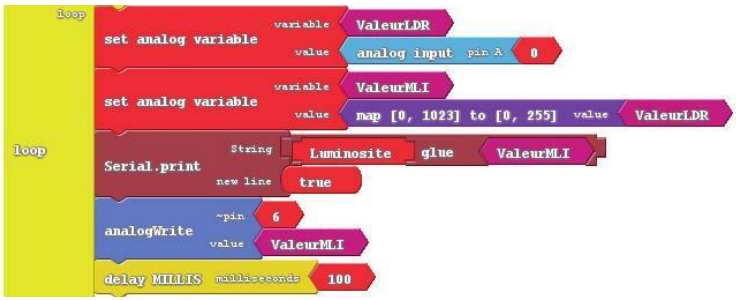


Vous voyez ici aussi que la partie gauche affiche différentes catégories organisées dans des conteneurs de couleur. La partie droite affiche la zone de script où sont déposés et organisés les différents blocs. En guise de simple exemple d'introduction, nous devons déterminer la valeur de luminosité d'une photorésistance (LDR) en vue de piloter la luminosité d'une LED et d'éditer simultanément la valeur MLI dans le moniteur série. Le schéma des connexions correspondant se présente ainsi :

Figure 24-14 ▶
Le schéma des connexions pour interroger la luminosité via la photorésistance

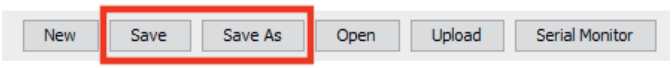


Passons maintenant à la conversion avec ArduBlock, qui se présente comme suit :

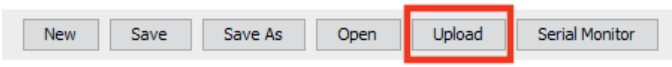


◀ Figure 24-15
Le script ArduBlock pour interroger la luminosité via la photorésistance

Après la construction du script, celui-ci doit d'abord être enregistré dans le système de fichiers au moyen des boutons *Save* :



Vous pourrez ensuite entreprendre le téléchargement sur la carte Arduino avec le bouton *Upload* :



Immédiatement après que vous avez cliqué sur ce bouton, vous pourrez voir le sketch généré par ArduBlock dans l'éditeur de l'environnement de développement Arduino, suivi par la compilation et le téléchargement. Admirez comment s'opère la conversion de la structure par blocs en langage C++ !

Open Roberta Lab

Une initiative de l'Institut Fraunhofer a donné un autre outil intéressant : Open Roberta Lab. Ce projet basé sur un navigateur a été développé dans le but de permettre à des enfants et à des adolescents de contrôler et de commander des systèmes robotisés sans avoir de connaissances de la programmation. Il est possible de commander en plus de la carte Arduino la minicarte à but pédagogique *micro:bit* ainsi que *LEGO Mindstorms* et le robot *mBot* du fabricant chinois Makeblock.

Le navigateur emploie un dérivé de Scratch, appelé NEPO. Comme dans S4A ou ArduBlock, la programmation s'effectue à l'aide de composants visuels. Vous pouvez également programmer vos propres composants NEPO et les insérer dans vos scripts. L'intérêt majeur dans l'environnement de déve-

loppement Open Roberta Lab est la prise en charge d'autres langages de programmation. Ainsi, vous pouvez voir sous forme de sketch Arduino le script que vous avez compilé à partir de composants. La figure 24-16 montre comment un simple script pour faire clignoter a été démarré dans Open Roberta Lab. Vous pouvez représenter dans la fenêtre de droite le code de programmation comme il apparaît sous forme de sketch Arduino :

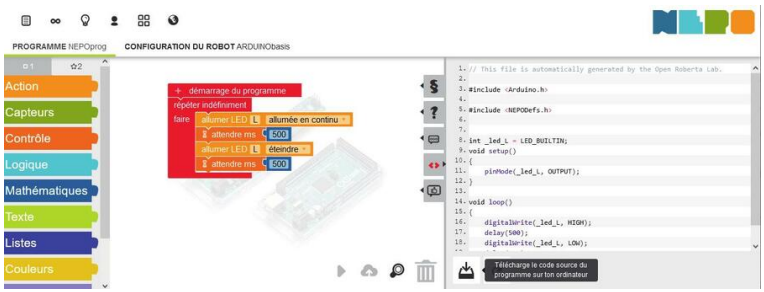


Figure 24-16 ▶
Le script peut également être représenté dans Open Roberta sous forme de sketch Arduino.

L'environnement de développement avec le navigateur Open Roberta Lab est accessible via le lien ci-après. Pour plus de commodité, un menu de sélection s'affiche dès le début, à partir duquel vous choisissez l'appareil que vous souhaitez programmer.

https://lab.open-roberta.org/

Node-RED, langage de programmation par blocs pour l'IoT

J'en arrive maintenant à un environnement de développement qui a tout pour plaire au sens propre du terme : *Node-RED*. Node-RED est un environnement de développement graphique développé par IBM, qui permet de réaliser des applications dans le domaine de l'Internet des objets (IoT) à l'aide d'un principe modulaire simple, similaire à l'environnement Scratch. Node-RED est devenu une infrastructure intéressante pour réaliser des applications complexes qui vont bien au-delà du domaine amateur. De nombreuses sociétés développant des produits IoT misent sur cet environnement de développement. Il existe des interfaces pour les langages script et les langages de programmation.

La manipulation correspond à l'approche que l'on trouve dans Scratch, où des blocs fonctionnels individuels sont assemblés dans un environnement de script. Le site Internet pour *Node-RED* est :

https://nodered.org/

L'installation peut s'effectuer sur différentes plateformes et un Raspberry Pi est certainement une très bonne solution. J'aimerais pourtant commencer ici avec une installation basée sur Windows. Vous trouverez toutes les informations sur l'installation à l'adresse suivante :

https://nodered.org/docs/getting-started/windows#running-on-windows

Je vous renvoie ici au montage n° 26, dans lequel j'explique de manière approfondie Node-RED.

Problèmes courants

Les sources d'erreurs avec les langages de programmation par blocs peuvent être très variées. Si vous rencontrez des problèmes pour réaliser les applications que je vous propose dans ce montage, suivez les étapes ci-dessous.

- Essayez de trouver des indices pour savoir si l'erreur se situe plutôt au niveau du logiciel ou du matériel. Essayez de délimiter l'erreur.
- Le téléchargement du firmware a-t-il été bien accepté sur la carte Arduino ?
- Avez-vous entré le bon port COM ?
- Votre pare-feu bloque-t-il une connexion ?

Qu'avez-vous appris ?

Voilà encore un montage qui n'a pas été très amusant sur le plan manuel, mais qui, je l'espère, vous a permis d'en apprendre davantage. Si vous aviez du mal avec la programmation, les langages par blocs vous apporteront peut-être de nouvelles et captivantes possibilités pour contrôler votre carte Arduino.

- Dans ce montage, vous avez découvert les langages de programmation par blocs S4A, ArduBlock et Open Roberta Lab, qui sont des dérivés de Scratch.
- Vous vous êtes un peu familiarisé avec la puissante infrastructure IoT Node-RED.
- Le principe de fonctionnement d'un langage de programmation par blocs a été expliqué.
- Vous savez maintenant comment établir la connexion entre un langage de programmation par blocs et votre carte Arduino.

Quand Arduino rencontre Raspberry Pi

Raspberry Pi est un ordinateur monocarte bon marché sur lequel il est possible d'installer des systèmes d'exploitation comme Linux et Microsofts Windows 10 IoT Core. L'évolution poursuivant inlassablement sa course, je peux aujourd'hui avancer, en l'état actuel de mes connaissances, que plus d'une trentaine de systèmes d'exploitation peuvent fonctionner avec Raspberry Pi. À partir de la combinaison d'un Raspberry Pi comme serveur et d'une carte Arduino comme dispositif de commande, de mesure et de collecte de données, il est possible de développer des applications surprenantes. C'est ce que nous allons voir ici.

Réveillons l'Arduino qui sommeille dans tout Raspberry Pi

Certes, le sujet de ce livre n'est pas Raspberry Pi, mais il me paraît opportun d'étudier la carte d'un peu plus près afin d'en souligner les points forts. Raspberry Pi est un ordinateur monocarte de la taille d'une carte bancaire qui a été développé par la fondation britannique Raspberry Pi. Il coûte environ 35 €, ce qui est très abordable. Mais ce n'est pas pour cette raison que les gens manifestent de l'intérêt pour ce nano-ordinateur. Cet ordinateur – puisqu'il s'agit bien d'un ordinateur à part entière – réunit tout ce dont une personne a besoin pour s'aventurer dans l'univers de l'informatique et de la programmation. Il possède un processeur Broadcom BCM2837 de type ARM Cortex-A53 64 bits Quad Core cadencé à 1,2 GHz, une mémoire vive de 1 024 Mo, un port Ethernet pour le raccordement à un réseau, ainsi que quatre ports USB, Bluetooth 4.1 + LE et un port GPIO. Comme support de mémoire, le nano-ordinateur utilise une carte SD, une clé USB, voire un disque SSD.

Figure 25-1 ▶
Le Raspberry Pi

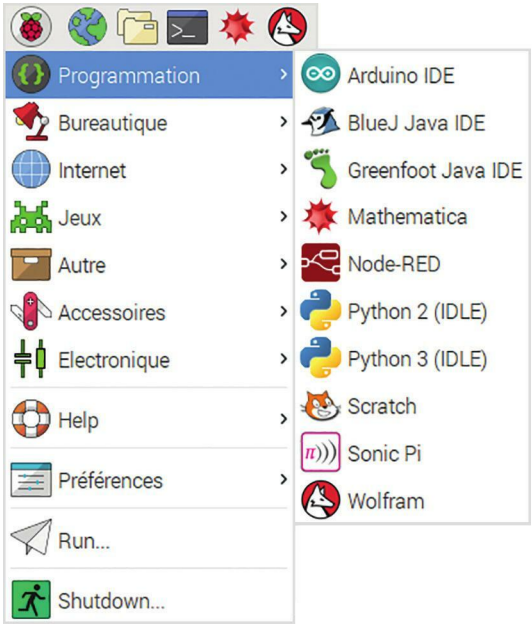
Si l'on veut raccorder un clavier et une souris, il est préférable d'utiliser un modèle sans fil avec dongle USB afin que les deux périphériques n'occupent qu'un seul port USB. Comme la majorité des écrans TFT ou des écrans plats disposent aujourd'hui d'une connexion HDMI, vous pouvez donc facilement les raccorder au port HDMI de type A *(full size)* de la carte Raspberry Pi. Si votre écran est un modèle plus ancien, sachez qu'il existe aussi des adaptateurs HDMI-DVI pour convertir le signal envoyé sur un port DVI. Voyons maintenant les préparatifs nécessaires en vue du raccordement d'une carte Arduino Uno à un Raspberry Pi afin que les deux systèmes puissent échanger des informations.

Installation de l'IDE Arduino sur le Raspberry Pi

Vous pouvez évidemment continuer à programmer votre carte Arduino Uno dans votre environnement de développement habituel depuis votre ordinateur sous Windows, Mac ou Linux. Mais, comme tôt ou tard, nous allons raccorder les deux cartes, je vais vous montrer ici comment le faire directement depuis le Raspberry Pi. En effet, qu'est-ce qui vous empêche d'utiliser l'IDE Arduino sur le Raspberry Pi ?! Ouvrez une fenêtre de terminal sur le Raspberry Pi et saisissez les deux lignes suivantes :

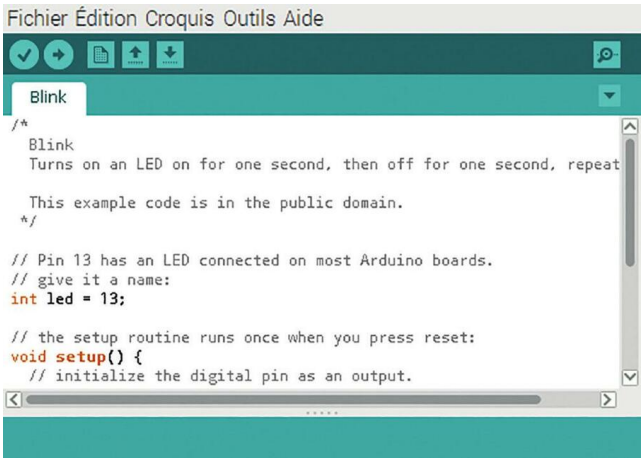
```
# sudo apt-get update
# sudo apt-get install arduino
```

Pour l'installation, nous utiliserons le gestionnaire de paquets APT *(Advanced Packaging Tool)*. La première ligne actualise les listes de paquets et la deuxième installe l'IDE Arduino. Une fois l'installation réussie, une nouvelle commande apparaît dans le menu Développement : il s'agit de l'IDE d'Arduino.



◀ Figure 25-2
L'IDE Arduino dans le menu Développement

Lorsque vous démarrez l'IDE à l'aide de cette commande, la fenêtre bien connue s'ouvre après quelques instants. Vous devez encore choisir le port série correct. Sous Linux, il s'agit de */dev/ttyACM0* pour la carte Uno. Les modèles plus anciens utilisent */dev/ttyUSB0*.



◀ Figure 25-3
Fenêtre 1.0.5 de l'environnement de développement

Vous pouvez vérifier que tout à bien fonctionné en lançant le sketch Blink que j'ai également chargé. Revenons-en à la communication entre l'Arduino et le Raspberry Pi. Nous allons utiliser ici Firmata, un protocole de

communication entre ordinateurs ou programmes. Voyons maintenant comment établir une connexion entre l'Arduino et le Raspberry Pi afin que les deux cartes puissent échanger des données.

Firmata

Côté Arduino, nous disposons déjà de tout le nécessaire. Firmata fait partie de l'environnement de développement Arduino et il suffit de le charger en tant que firmware sous la forme d'un sketch. Côté Raspberry Pi, ce n'est pas tout à fait la même chose. Le langage habituel du Raspberry Pi, Python, fait déjà partie de la distribution Linux Raspbian Jessie, que je recommande aux débutants. Il nous faut donc uniquement installer Firmata. Le paquet Python correspondant se nomme pyFirmata.

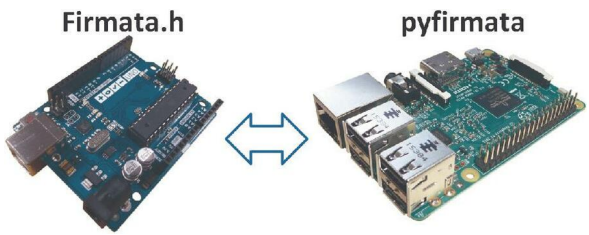


Figure 25-4 ▶
Firmata sur l'Arduino et le Raspberry Pi

Comme Firmata fait déjà partie de l'environnement de développement Arduino, il suffit de s'assurer de son bon fonctionnement en ajoutant le fichier d'en-tête Firmata.h. Côté Raspberry Pi, nous avons quelques bricoles à installer. Saisissez la commande suivante dans le terminal :

```
# sudo pip install pyfirmata
```

Nous avons besoin d'un autre paquet pour Python qui s'intitule pySerial et qui est chargé de la communication avec le port série. Il est déjà inclus dans la dernière version du système d'exploitation Raspbian Jessie. Tout est maintenant prêt pour procéder au premier essai. Nous allons dire bonjour à Arduino et nous ferons clignoter la LED 13 du Raspberry Pi. Comment allons-nous faire ?

Préparation de l'Arduino

Vous devez évidemment munir l'Arduino du firmware de Firmata en chargeant le sketch correspondant et en le téléversant sur la carte. Activez la commande *Fichiers* | *Exemples* | *Firmata* | *StandardFirmata* pour ouvrir le sketch dans l'environnement de développement, puis téléversez-le sur la carte Arduino. Vous n'avez rien de plus à faire pour l'instant.

Préparation du Raspberry Pi

Comme nous travaillerons avec le langage de programmation Python sur le Raspberry Pi, il est conseillé d'installer un environnement de développement Python. Actuellement, j'utilise SPE (Stani's Python Editor) que vous pouvez installer à l'aide des lignes suivantes :

```
# sodo apt-get update
# sudo apt-get install spe
```

Toutefois, cet environnement de développement ralentit le Raspberry Pi, comme vous pourrez le constater en jetant un œil à l'indicateur de performances du processeur dans la barre des tâches. Au lieu de SPE, vous pouvez aussi utiliser le simple éditeur de texte Nano qui s'installe à l'aide de la ligne de commande suivante :

```
# sudo apt-get install nano
```

Pour ouvrir l'éditeur, il suffit ensuite de saisir la commande suivante dans le terminal :

```
# nano
```

Nous allons néanmoins continuer avec SPE. Après son installation, vous le trouverez dans le dossier Développement du menu Démarrer de Linux :

◀ Figure 25-5
Ouverture de Stani's Python Editor

Vous pouvez saisir le code suivant dans l'éditeur pour faire clignoter la LED 13. Ne vous inquiétez pas, nous n'en resterons pas là ! Nous ferons aussi des choses un peu plus compliquées. Cet exercice nous permet simplement de nous mettre en jambes :

```
#!/usr/bin/python
# -*- encoding: utf8 -*-
from time import sleep   # import de sleep
from pyfirmata import Arduino, util  # Import de Arduino, util

# Arduino-board communication via le port série
arduinoboard = Arduino('/dev/ttyACM0')

#Programmation des broches Arduino
pin13 = arduinoboard.get_pin('d:13:o')
while True:
    pin13.write(1)  # LED allumée
    sleep(1)        # Pause de 1 seconde
    pin13.write(0)  # LED éteinte
    sleep(1)        # Pause de 1 seconde
```

Examinons la signification des différentes lignes du code.

Ligne 1

Afin que le script Python soit correctement détecté par l'interpréteur, la première ligne commence sur les systèmes Linux par les caractères #! suivis du chemin d'accès absolu de l'incontournable interpréteur.

```
#!/usr/bin/python
```

Cette ligne se nomme le *shebang*. L'emplacement de l'interpréteur Python varie selon les systèmes Linux. Par conséquent, la ligne telle qu'elle est présentée ici n'est pas universellement valable. Il existe une meilleure variante qui utilise le programme `env`.

```
#!/usr/bin/env python
```

Le code `env` charge les variables d'environnement standard de la configuration du système d'exploitation qui prend aussi en charge la variable d'environnement `PATH`. Sur mon ordinateur, l'interpréteur Python se trouve sous `/usr/bin/python`.

Ligne 2

Pour pouvoir utiliser des caractères accentués (même dans les commentaires précédés par #), il est nécessaire de préciser l'encodage des caractères UTF-8 par cette ligne.

```
# -*- encoding: utf8 -*-
```

Ligne 3

Comme nous avons besoin de la fonction `sleep` pour insérer une pause, celle-ci est importée à la ligne 3 par l'instruction `from/import`.

Ligne 4

Pour pouvoir utiliser la multitude de fonctions de pyFirmata après son installation, le paquet est inséré dans le code à la ligne 4 par l'instruction `from/import`.

Ligne 7

Comme nous voulons communiquer et contrôler la carte Arduino via le port série, nous devons nous y connecter. Il s'agit du même *device* (appareil) */dev/ttyACM0* que celui que nous avons déjà utilisé pour programmer la carte Arduino. Pour que Python puisse avoir accès au port série, nous avions précédemment installé le paquet *python-serial* pendant la phase préparatoire.

```
arduinoboard =Arduino('/dev/ttyACM0')
```

Par cette ligne, pyFirmata crée une instance de *pyfirmata* que nous avons nommée *arduinoboard*. Comme argument, nous transmettons précisément le nom d'appareil que je viens d'indiquer.

Ligne 10

Les différentes broches de la carte peuvent être contrôlées ou configurées par la méthode `get_pin`. La configuration s'effectue par le biais d'une séquence de caractères au format suivant :

```
(a|d:<PinNr>:i|o|p|s)
```

Les trois informations nécessaires sont énumérées en étant séparées par deux-points. En voici la signification :

- L'instruction est-elle destinée à une broche analogique (`a`) ou numérique (`d`) ?
- De quelle broche s'agit-il (`PinNr`) ?
- Quel mode faut-il utiliser (`i` : entrée, `o` : sortie, `p` : MLI, `s` : servo) ?

C'est donc ce que nous faisons à la ligne 10.

```
pin13 = arduinoboard.get_pin('d:13:o')
```

Une variable intitulée `pin13` est initialisée à l'aide de la méthode `get_pin` afin de nous permettre de manipuler cette broche (numérique, 13, sortie) conformément aux instructions transmises aux lignes 11 à 15.

Lignes 11 à 15

Une boucle `while` permet d'exécuter en continu les instructions énoncées dans le corps de la boucle :

```
while True:
    pin13.write(1) # LED allumée
    sleep(1)       # Pause de 1 seconde
    pin13.write(0) # LED éteinte
    sleep(1)       # Pause de 1 seconde
```

Pour associer différents niveaux à la LED qui est connectée à la broche 13, nous utilisons la méthode `write` avec l'argument `1` pour le niveau HIGH ou `0` pour le niveau LOW. Entre les deux, nous plaçons la fonction `sleep` qui interrompt l'exécution du programme pendant une durée d'une seconde. La boucle `while` est exécutée tant que l'instruction suivante est vraie (`True`). Nous n'avons pas employé de variable comme instruction, mais la constante `True`, ce qui signifie que la boucle est exécutée à l'infini ou jusqu'à ce que l'utilisateur interrompe manuellement l'exécution du script à l'aide du raccourci Ctrl+C. Si, après avoir téléversé le sketch firmata depuis les exemples de l'IDE, vous avez démarré le script depuis le Raspberry Pi, observez la LED RX de la carte Arduino. Elle s'allume et s'éteint avec un intervalle d'une seconde. On peut donc en conclure que des informations sont transmises depuis le Raspberry Pi à l'Arduino via le port série avec le même intervalle pour contrôler la LED.

Commande par MLI

Nous allons maintenant voir comment commander une LED raccordée à l'une des broches MLI à l'aide d'un signal MLI. Nous en avons déjà vu le principe dans le montage n° 1. Je voudrais ouvrir une interface graphique sur le Raspberry Pi afin d'envoyer le signal MLI à la carte Arduino à l'aide d'un potentiomètre linéaire comme celui illustré ci-dessous.

Le potentiomètre linéaire permet de choisir des valeurs comprises entre 0 et 100 qui correspondent aux pourcentages de MLI. Attention toutefois, car la fonction `write` de pyFirmata, qui s'occupe de générer le signal MLI accepte des valeurs comprises entre 0,0 et 1,0. J'ai donc intentionnellement employé des valeurs à virgule flottante, car la fonction attend des valeurs de type `float`. Examinons le script Python de plus près. Python ne peut afficher de but en blanc des éléments graphiques, comme des boutons, étiquettes,

des potentiomètres linéaires ou autres. Pour ce faire, nous utilisons une bibliothèque nommée Tkinter. Qu'est-ce donc ? Il s'agit de la première boîte à outils d'interface graphique pour Python. Elle permet de créer sous Python des programmes ayant une interface graphique. La boîte à outils Tk a initialement été développée pour le langage Tcl *(Tool Command Language)*, mais entre temps, elle a pris place dans la bibliothèque standard de Python. Le module Tkinter *(Tk-Interface)* permet à l'utilisateur de programmer très facilement des applications Tk sans devoir installer au préalable des logiciels ou des bibliothèques supplémentaires. Examinons le script Python que j'ai divisé en blocs pour une meilleure lisibilité.

Initialisation

Pendant la phase d'initialisation, nous importerons aussi bien pyfirmata que Tkinter. Nous utilisons encore arduinoboard, comme dans l'exemple précédent. La broche 3, sur laquelle la diode est raccordée, doit être initialisée en tant que sortie MLI, ce que nous faisons à l'aide du mode `p`.

```
1  #!usr/bin/env python
2  # *-* encoding: utf8 *-*
3  import pyfirmata
4  from Tkinter import *
5
6  arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')
7
8  pin3 = arduinoboard.get_pin('d:3:p') # Broche 3 en MLI
```

Fonctions requises

Les fonctions `cleanup` et `setPWM` sont utilisées lors de l'exécution du script : on s'en serait douté. La fonction `cleanup` est exécutée au moment où vous fermez l'interface graphique en cliquant sur la croix dans le coin supérieur droit. Elle fait en sorte que la LED soit éteinte au moyen de la fonction `write`. La fonction `setPWM` commande la LED et elle est exécutée tant que vous modifiez la position du curseur du potentiomètre.

```
10 def cleanup():
11     # LED 3 eteinte
12         pin3.write(0)
13         arduinoboard.exit()
14
15 def setPWM(pwm):
16     # LED 3 pilotée en PWM
17         # Les valeurs comprises entre 0 et 1 sont acceptées
18         pin3.write(float(pwm)/100.0)
```

La fonction `write` de la commande par MLI attend des valeurs comprises entre 0 et 1, ce qui signifie que l'argument doit être de type `float`. Comme le potentiomètre transmettra par la suite, en réponse au déplacement du curseur, une valeur de paramètre `pwm` comprise entre 0 et 100, cette valeur doit donc être divisée par 100.

Préparation de l'interface graphique GUI et du potentiomètre linéaire

L'interface graphique est initialisée à la ligne 20, qui prépare plus précisément l'instance `master` de la boîte de dialogue `wm_protocol`, ferme la fenêtre en effaçant son contenu, opération qui s'achève par l'exécution de la fonction `cleanup` qui éteint la diode. À la ligne 22, `wm_title` permet de nommer l'application, nom qui sera affiché dans la barre de titre. Le potentiomètre est initialisé avec les valeurs correspondantes et son exécution démarre à la ligne 24. `from_` et `to` définissent la plage de valeurs transmises par le potentiomètre. Quand le curseur est actionné, il faut afficher immédiatement le résultat, ce qui est assuré par l'instruction `command` suivie de la fonction exécutée. L'orientation est définie par `orient` ; ici, elle est horizontale. Ensuite, nous précisons encore la longueur (`length`) et nous affichons le nom du potentiomètre au moyen d'une étiquette (`label`).

```
20  #GUI    Interface graphique
21  master = Tk()
22  master.wm_protocol("WM_DELETE_WINDOW", cleanup)
23  master.wm_title('Contrôle PWM')
24
25  # Initialisation du potentiomètre linéaire
26  scale = Scale(master,
27      from_ = 0,
28      to = 100,
29      command = setPWM,
30      orient = HORIZONTAL,
31      length = 400,
32      label = 'Valeur PWM')
33  scale.set(50) # Curseur au centre
```

Démarrage du programme

Le gestionnaire d'interface se sert de `pack` pour centrer le potentiomètre à l'intérieur de la boîte de dialogue. L'activation de `mainloop` affiche la boîte de dialogue et maintient le programme dans une boucle sans fin, en attendant que se produisent des événements intéressants comme le déplacement du curseur qui déclenche l'action programmée.

```
35  scale.pack(anchor = CENTER) # Placer au milieu
36  master.mainloop() # Démarrage de la boucle d'interrogation Tk
```

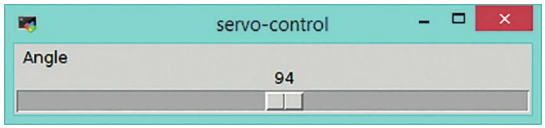
Il n'y a pas grand-chose à ajouter sur ce script assez simple. Tkinter est bien plus performant que ce que j'ai pu montrer dans ces quelques pages et je ne peux que vous conseiller la lecture d'ouvrages spécialisés ou la consultation de ressources sur Internet. Comme vous vous êtes maintenant familiarisé avec le fonctionnement d'une broche MLI, vous allez pouvoir commander un servomoteur au moyen d'un potentiomètre.

ATTENTION !

Avec la version 3 de Python, vérifiez que vous écrivez bien `tkinter` (avec un `t` minuscule) lors de son importation. La bibliothèque a été renommée.

Commande d'un servomoteur

Nous avons vu précédemment comment commander un servomoteur. Ici, vous apprendrez à régler précisément l'angle du servomoteur à l'aide d'un potentiomètre.



Pour en savoir davantage sur le brochage d'un servomoteur, je vous invite à relire le montage n° 16, « Le moteur pas-à-pas ». J'aimerais commander le moteur au moyen de la broche MLI 3, mais il est aussi possible d'utiliser une broche qui ne soit pas dotée de cette fonctionnalité de modulation, comme la broche 7. Examinons le code Python qui est très proche de celui de l'exemple précédent. Comme vous pourrez le constater, une fois que l'on a compris le principe de base, il suffit parfois de modifier un détail pour accéder à d'autres fonctionnalités.

```
1  #!usr/bin/env python
2  # *-* encoding: utf8 *-*
3  import pyfirmata
4  from Tkinter import *
5
6  arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')
7
8  it = pyfirmata.util.Iterator(arduinoboard)
9  it.start
10
11 pin3 = arduinoboard.get_pin('d:3:s') # Broche 3 en mode servo
```

La seule différence dans ce segment de code réside dans le changement de mode qui passe à `s` pour la commande d'un servomoteur. Les deux fonctions suivantes restent quasiment inchangées :

```
13 def cleanup():
14     # Broche 3 désactivée
15     pin3.write(0)
16     arduinoboard.exit()
17
18 def moveServo(a):
19     # La broche 3 est en mode Servo
20     pin3.write(a)
```

La valeur du potentiomètre est ici aussi transmise au paramètre de la fonction `moveServo` pour commander le moteur à l'aide de la méthode `write`. L'interface du programme est créée de la même façon que précédemment et je me suis contenté d'en changer le nom :

```
# GUI Interface graphique
master = Tk()
master.wm_protocol("WM_DELETE_WINDOW", cleanup)
master.wm_title('servo-control')
```

Nous en arrivons à l'initialisation du potentiomètre qui doit transmettre des valeurs comprises entre 0 et 179 à la fonction `moveServo`. Ensuite, le script démarre avec `mainloop`.

```
# Initialisation du potentiomètre linéaire
scale = Scale(master,
    from_ = 0,
    to = 179,
    command = moveServo,
    orient = HORIZONTAL,
    length = 400,
    label = 'Angle')

scale.pack(anchor = CENTER) # Placer au milieu
mainloop() # Démarrage de la boucle d'interrogation Tk
```

Interrogation d'un bouton-poussoir

Jusqu'ici, nous avons toujours transmis des informations à la carte Arduino. Nous allons maintenant faire l'inverse en interrogeant un bouton-poussoir qui est raccordé à la broche 8. Comme procède-t-on avec pyFirmata ? Le code est assez évident :

```
#!/usr/bin/env python
# *-* encoding: utf8 *-*
import pyfirmata
arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')

pin8 = arduinoboard.get_pin('d:8:i') # Broche digitale 8 en mode INPUT

it = pyfirmata.util.Iterator(arduinoboard)
it.start()
pin8.enable_reporting()

while True:
    pin8_state = pin8.read() # Lecture de l'état
    if pin8_state == True:
        print 'Poussoir appuyé'
    if pin8_state == False:
        print 'poussoir relaché'
    arduinoboard.pass_time(0.5) # Pause
```

Ligne 6

Pour pouvoir interroger l'état d'un bouton-poussoir connecté sur une broche, nous devons évidemment la programmer en tant qu'entrée, ce que nous faisons de ce pas avec `d:8:i` (*i* correspond à *input* ou à entrée).

Lignes 8 et 9

Pour lire une broche d'entrée sous pyFirmata, vous ne pouvez pas tout simplement activer une fonction `read`, comme nous le ferons à la ligne 13. Il faut implémenter un *Iterator Thread* qui veille à ce que les broches de la carte Arduino communiquent la valeur courante lors de leur interrogation. Cela évite aussi que ne se produise un *buffer overflow* qui bloquerait toute la communication sur le port série.

Ligne 10

Par `enable_reporting`, vous indiquez à pyFirmata que vous voulez surveiller la broche.

Lignes 12 et 13

La boucle `while` interroge l'état de la broche 8 en continu en utilisant la méthode `read`.

Lignes 14 à 17

Les instructions `if` interrogent l'état `True` qui correspond au bouton enfoncé et l'état `False` qui correspond au bouton non enfoncé et affichent le résultat par l'instruction `print`.

Ligne 18

La méthode `pass_time` prévoit une pause d'une demi-seconde après chaque affichage.

Vous constaterez qu'au démarrage du script, rien n'indique que le bouton n'est pas enfoncé. Pourquoi le message correspondant ne s'affiche-t-il pas ? En fait, quand le bouton n'a pas encore été enfoncé, l'état correspond à `None`. Libre à vous de compléter le code afin que cet état soit aussi interrogé et qu'un message s'affiche.

Interrogation d'un port analogique

Pour finir, nous allons voir comment interroger une entrée analogique et les aspects à prendre en considération. Là encore, le script est assez évident :

```
1  #!/usr/bin/env python
2  # *-* encoding: utf8 *-*
3  import pyfirmata
4  from time import sleep  # import de sleep
5  arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')
6
7  pin0 = arduinoboard.get_pin('a:0:i') # Broche 0 en entrée analogique
8
9  it = pyfirmata.util.Iterator(arduinoboard)
10 it.start()
11 pin0.enable_reporting()
12
13 while True:
14     value = pin0.read() # Lire la valeur analogique
15     print value         # Affichage de la valeur
16     sleep(1)            # Pause 1 seconde
```

Ligne 7

La broche analogique 0 est désignée par un `a` (pour `analogique`) et elle est programmée en tant qu'entrée par le biais du mode `i`.

Lignes 13 à 16

La valeur analogique est lue sur la broche 0 à l'intérieur de la boucle `while` au moyen de la méthode `read`. Le résultat est compris entre 0,0 et 1,0. Ensuite, le programme marque une courte pause d'1 seconde.

Lorsque vous faites lentement tourner le potentiomètre de la gauche vers la droite, il se peut que vous obteniez les valeurs suivantes :

```
pi@raspberrypiJB:~ $ python analog0.py
None
0.0
0.7019
0.829
0.8563
0.914
1.0
```

Au tout début de l'affichage, on peut lire `None`, ce qui signifie qu'aucune valeur valide n'a pu être lue. Cela se produit de temps en temps au début de l'interrogation. Voici comment l'éviter :

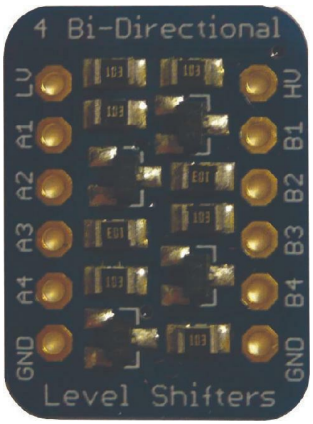
```
13 while True:
14     value = pin0.read()             # Lire la valeur analogique
15     if value != None:
16         print 'Valeur: %f' % value  # Affichage de la valeur
17         sleep(1)                    # Pause 1 seconde
```

À la ligne 15, j'ai placé une instruction `if` qui intercepte la valeur `None`, le cas échéant. Pour améliorer la présentation, vous pouvez aussi précéder l'affichage d'un texte ou de la mention du type de données, comme à la ligne 16. Vous obtenez alors le code suivant :

```
pi@raspberrypiJB:~ $ python analog.py
Valeur 0.000000
Valeur 0.012700
Valeur 0.326500
Valeur 0.770300
Valeur 0.849500
Valeur 1.000000
```

Liaison série entre le Raspberry Pi et l'Arduino

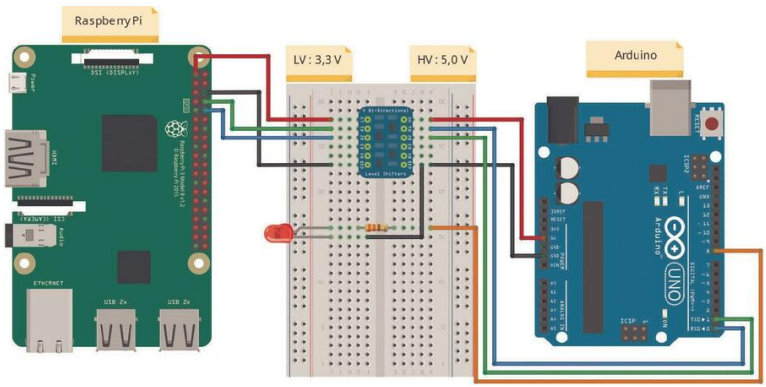
Jusqu'ici, les informations échangées entre le Raspberry Pi et l'Arduino ont transité par la liaison USB qui a servi à l'acheminement des données série. Mais il est aussi possible d'établir directement une liaison TTL série à condition de respecter la précaution suivante : le Raspberry Pi fonctionne avec une tension d'alimentation maximale sur ses entrées et sorties GPIO de 3,3 V, tandis que l'Arduino utilise une tension de 5 V. Si vous raccordez les deux cartes l'une à l'autre sans prendre de précautions, c'est la loi du plus fort qui prévaut et l'un des protagonistes restera sur le carreau. Le Raspberry Pi recevra une tension trop élevée. Et comme ses broches GPIO sont directement reliées au processeur, sans protection, celui-ci grillera et la carte sera bonne à jeter. Ce n'est donc pas une bonne idée ! Comment l'éviter ? Nous pourrions utiliser un diviseur de tension afin de nous assurer que le Raspberry Pi reçoit bien une tension de 3,3 V. J'ai préféré employer un composant meilleur marché qui s'appelle un convertisseur logique ou *levelshifter*, comme le modèle proposé par Adafruit.



◀ Figure 25-6
Convertisseur logique Adafruit

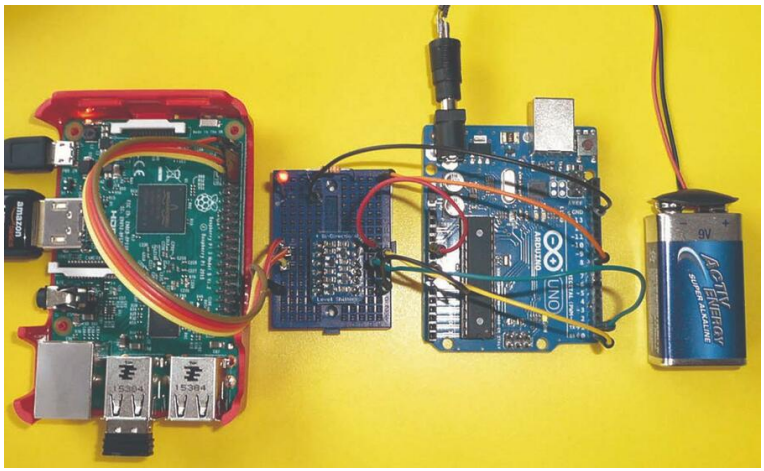
Ce composant permet non seulement de convertir la tension pour les liaisons RX/TX, mais aussi pour les bus I²C et SPI. Sur le côté gauche se trouvent les broches de raccordement en basse tension (LV ou *LOW Voltage*) et sur le côté droit, il y a les broches haute tension (HV ou *HIGH Voltage*). Voyons maintenant comment raccorder correctement le Raspberry Pi et l'Arduino pour que les deux cartes réussissent à communiquer via les ports série.

Figure 25-7 ▶
Le convertisseur logique sert d'intermédiaire entre le Raspberry Pi et l'Arduino.



Il va sans dire que les lignes émettrices et réceptrices des deux cartes doivent être croisées, car si vous voulez que la liaison TX verte, par laquelle le Raspberry Pi transmet des données, réussisse à se faire entendre par l'Arduino, cette dernière doit être connectée à sa liaison réceptrice RX bleue, et inversement. Le montage suivant permet de faire clignoter une LED connectée à la broche 8 par le biais des deux liaisons RX/TX.

Figure 25-8 ▶
Le Raspberry Pi commande l'Arduino via les liaisons RX/TX.



Vous pouvez constater qu'il n'y a pas de connexion USB entre les deux cartes et que la communication passe exclusivement par l'interface UART. L'alimentation électrique de l'Arduino s'effectue au moyen d'une pile de 9 V. Quelles conditions doivent être remplies pour que cette forme de communication fonctionne ? Tout d'abord, je dois vous en dire davantage sur le port série du Raspberry Pi. Par défaut, le Raspberry Pi utilise le port série en tant qu'interface de console par l'intermédiaire duquel vous pouvez accéder au système depuis l'extérieur, même si vous n'y avez pas raccordé de moniteur, de souris ou de clavier – quasi *headless*. Mais ce moyen d'accès dont nous n'avons pas absolument besoin pour le moment bloque les broches RX/TX et nous empêche de poursuivre notre expérience. Nous devons donc couper ce lien. La solution réside dans la configuration du port série. Ouvrez la configuration du Raspberry Pi et affichez l'onglet *Interfaces*.

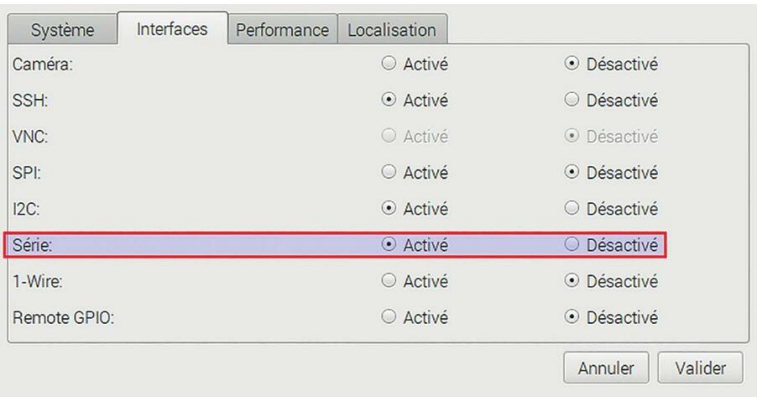


Figure 25-9
Configuration du port série du Raspberry Pi

Activez le port série à l'aide de l'option correspondante de la boîte de dialogue, puis redémarrez le système. Ensuite, il ne vous reste plus qu'à apporter quelques modifications au fameux script Blink.

```
#!/usr/bin/env python
# -*- encoding: utf8 -*-
from time import sleep              # import de sleep
from pyfirmata import Arduino, util # Import des utilitaires

# Adressage de la carte Arduino via le port série
arduinoboard = Arduino('/dev/ttyS0')

# Programmation des broches Arduino
pin13 = arduinoboard.get_pin('d:13:o')
while True:
    pin13.write(1)  # LED allumée
    sleep(1)        # Pause de 1 seconde
    pin13.write(0)  # LED éteinte
    sleep(1)        # Pause 1 seconde
```

À première vue, le code ne semble pas avoir changé. Pourtant, il y a une différence majeure. Regardez attentivement la ligne 7 à laquelle le port série est initialisé.

Avant :

```
arduinoboard = Arduino('/dev/ttyACM0')
```

Après :

```
arduinoboard = Arduino('/dev/ttyS0')
```

Le *device* que nous avions utilisé précédemment se rapportait à l'interface USB. Après la correction, le *device* accède à l'interface UART.

Qu'avez-vous appris ?

- Vous avez vu comment programmer votre Arduino Uno avec le Raspberry Pi au moyen de l'IDE Arduino qui y est installé.
- Vous vous êtes servi de Firmata pour établir et commander la communication entre le Raspberry Pi et l'Arduino Uno.
- Par ailleurs, vous avez constaté à quel point il est facile de créer des éléments d'interface utilisateur graphiques avec Tkinter afin de contrôler les saisies sans passer par des lignes de commande. Cela vous a permis d'envoyer des signaux MLI à l'Arduino Uno, de commander un servomoteur et d'interroger le port analogique.
- Enfin, vous avez raccordé votre carte Arduino et Raspberry Pi au moyen des broches du port série.

Montage 26

Programmer pour l'Internet des objets avec Node-RED

L'*Internet des objets* (*Internet of Things* en anglais, ou IoT) est l'interconnexion entre des objets (Things) du monde réel et des objets du monde virtuel (Internet). Il s'agit en somme de tout interconnecter pour nous simplifier la vie grâce à une communication qui permet d'échanger des données de toute sorte. Voilà pour la théorie. Imaginez que des capteurs disposés dans votre réfrigérateur détectent la quantité de produits qui s'y trouvent et à quel moment vous pourriez peut-être bientôt manquer de quelque chose. Autre exemple : votre système de surveillance prend alors les devants, commande sur Internet chez votre marchand en ligne préféré la quantité suffisante de produit avant qu'il ne vienne à manquer, et le produit est ensuite livré chez vous. Votre véhicule dispose d'un ordinateur de bord, qui prend rendez-vous automatiquement à l'avance dans votre concession lorsqu'une révision arrive à échéance ou qu'un quelconque problème survient. Il est aussi en mesure de commander le cas échéant les pièces de rechange nécessaires. Ce genre d'infrastructure interconnectée va évidemment de pair avec le risque d'une surveillance globale. En effet, les données ne sont pas soumises à l'heure d'aujourd'hui aux réglementations strictes sur la protection des données personnelles que l'on est en droit d'espérer en tant que personne privée. Nous n'approfondirons pas ici ce sujet explosif même s'il devient de plus en plus épineux avec l'Internet des objets.

Dans ce montage, vous allez découvrir le framework Node-RED, qui vous permettra de représenter des données que vous aurez générées à l'aide d'un capteur de température et d'humidité intégré à votre carte Arduino. Vous pourrez afficher les données de manière attrayante dans Node-RED via un

tableau de bord avant d'y accéder par une URL. Vous apprendrez également comment générer et envoyer automatiquement des e-mails à partir du framework. Enfin, vous verrez ce qu'est le format de fichiers JSON et à quoi il va nous servir.

Comment fonctionne Node-RED ?

J'aimerais vous présenter plus en détail dans ce montage l'outil de développement Node-RED conçu par IBM, dont nous avons déjà fait connaissance dans le montage n° 24 sur les langages de programmation par blocs. Au moyen d'une interface graphique, il est possible de créer en un rien de temps des connexions entre différents périphériques matériels ou instances via ce qu'on appellera des Flows (flux en français) en vue de réaliser un flux ou un échange d'informations.

Comme le langage de programmation Scratch, Node-RED fournit des blocs prédéfinis contenant des fonctions spécifiques, qui peuvent être combinés très facilement entre eux à partir d'un stock. Ces blocs, aussi appelés Nodes (nœuds en français), remplissent certaines tâches et constituent ensemble les flux précédemment mentionnés. La figure ci-après nous montre un flux simple, qui permet de faire clignoter une LED connectée à votre carte Arduino.

Figure 26-1 ▶
Un flux Node-RED simple

Voyons maintenant en détail comme tout cela fonctionne. Les Nodes sont organisés en différentes palettes et peuvent, comme avec Scratch, être placés sur l'espace de travail par un simple glisser-déposer. Node-RED s'appuie sur JavaScript et, côté serveur, sur la plateforme Node.js, qui sert à développer des logiciels pour des applications réseau. Comme Node-RED fonctionne au sein d'un navigateur web, l'application n'est pas tributaire d'une plateforme. Je n'ai constaté aucun problème lié à l'utilisation de Firefox, Opera, Chrome ou Internet Explorer. La figure ci-après montre les palettes de base. J'en ai ajouté déjà une nouvelle pour la commande de la carte Arduino. Nous verrons plus loin comment ajouter des palettes.

Figure 26-2
Les différentes palettes de Node-RED

La page de démarrage de Node-RED se trouve à l'adresse Internet suivante :

https://nodered.org/

Vous y trouverez également toutes les informations utiles.

Installation de Node-RED

Node-RED peut être installé localement sur un ordinateur ainsi que sur divers autres ordinateurs qui fonctionnent alors comme des serveurs Node-RED. Comme il s'agit d'une application de navigateur, il est très simple d'entrer l'URL correspondante suivie du port 1880 (par exemple : 192.168.178.58:1880) pour établir la connexion. Si vous devez utiliser l'extension Arduino, vous devez impérativement raccorder la carte Arduino au serveur et non à l'ordinateur qui fonctionne comme client. Tout utilisateur qui possède un Raspberry Pi peut donc y installer Node-RED très facilement. Les possibilités d'installation étant toujours susceptibles d'évoluer rapidement, mes explications sur le sujet risquent d'être dépassées. C'est pourquoi j'indique ici les sources, où sont très bien décrites les différentes étapes de l'installation. Je suis tout à fait conscient que tous les liens que je mentionne risquent un jour ou l'autre d'être obsolètes et de renvoyer à une page vide. Mais recourir à un moteur de recherche ne devrait pas poser trop de problèmes et devrait même permettre de franchir élégamment ce genre d'obstacle.

APERÇU RAPIDE AVEC FRED

Pour avoir un aperçu rapide et simple de Node-RED, nul besoin de l'installer sur votre ordinateur pour débuter et faire vos premiers pas. Il vous suffit d'utiliser FRED (Frontend pour Node-RED). Rendez-vous simplement sur la page Internet de FRED :

https://fred.sensetecnic.com/

Vous vous connectez, choisissez la version gratuite et quelques instants après, vous pouvez commencer à utiliser Node-RED. Il existe toutefois une restriction quant au nombre de nœuds pouvant être créés, restriction qui ne devrait pas constituer un gros problème. À vous de jouer. Amusez-vous bien !

Installation de Node-RED sur Windows

L'installation sur un ordinateur Windows est décrite à l'adresse Internet suivante :

https://nodered.org/docs/getting-started/windows

Installation de Node-RED sur un Raspberry Pi

L'installation sur un Raspberry Pi est décrite à l'adresse Internet suivante :

https://nodered.org/docs/getting-started/raspberrypi

Démarrage de Node-RED

Pour démarrer Node-RED sous Windows, allez dans le menu de démarrage et sélectionnez le programme *Node.js command prompt*, lequel ouvre la ligne de commande :

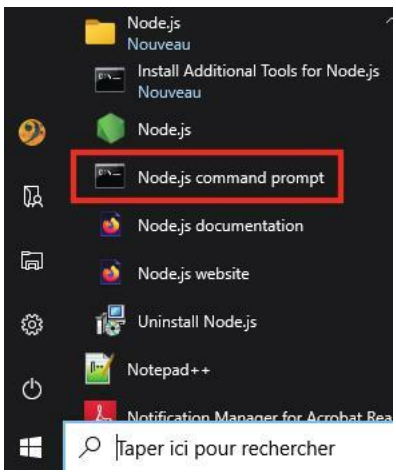


Figure 26-3 ▶ Démarrage de Node-RED sous Windows – ouvrir la ligne de commande

Une fois la ligne de commande ouverte, entrez la commande `node-red` et confirmez à l'aide de la touche Entrée :

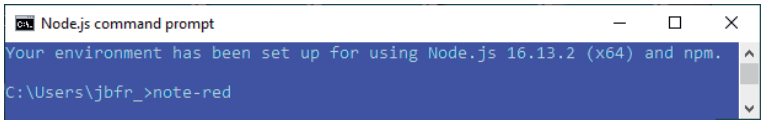


Figure 26-4
Démarrage de Node-RED sous Windows – entrer une commande

Un certain nombre de messages se déroulent ensuite. Le message le plus important pour vous se situe vers la fin :

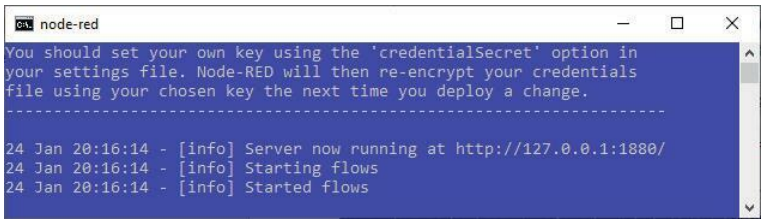


Figure 26-5
Messages de Node-RED après le démarrage

La dernière ligne indique que le serveur fonctionne localement sur votre ordinateur. Vous devez ensuite entrer l'adresse

```
http://127.0.0.1:1880
```

sous la forme d'une URL dans un navigateur. L'adresse IP 127.0.0.1 renvoie au localhost, autrement dit à l'ordinateur local. Le nombre 1880 à la fin de l'adresse IP indique le port ; il est séparé de l'adresse IP par deux points. Alternativement, vous pouvez également saisir l'adresse IP attribuée par le routeur, qui a été affectée à votre ordinateur dans votre réseau domestique.

Vous pouvez trouver celle-ci à l'aide de la commande `ipconfig` dans la ligne de commande :

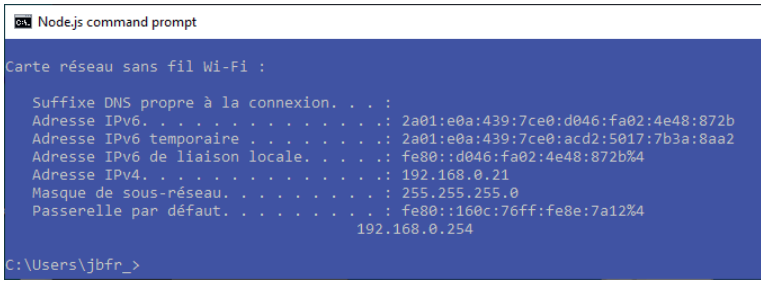


Figure 26-6
Trouver votre adresse IP

Dans mon cas, l'adresse IP est 192.168.178.21. Elle peut être adressée non seulement localement, mais aussi par tous les autres ordinateurs présents dans le réseau correspondant grâce à l'indication supplémentaire du port 1880. Le démarrage de Node-RED sur un Raspberry Pi s'opère de manière similaire. Vous devez aussi saisir dans une fenêtre de terminal la

commande `node-red`. Veillez à respecter les majuscules et les minuscules. J'ai résumé l'extrait suivant aux lignes les plus importantes :

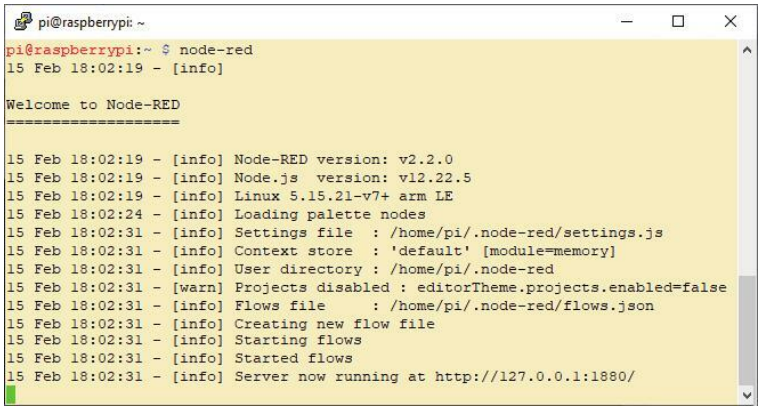


Figure 26-7 ▶ Démarrage de Node-RED sur un Raspberry Pi – entrer une commande

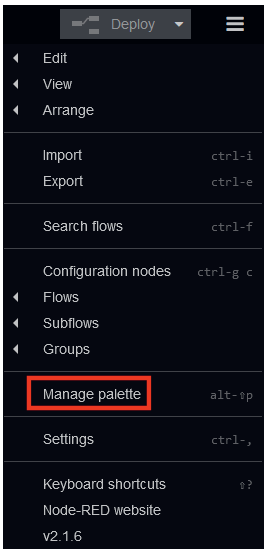
Là encore, vous pouvez trouver l'adresse IP réelle dans le réseau, mais ici la commande est `ifconfig` et vous devrez la lancer dans une fenêtre de terminal. Dans tous les cas, n'oubliez pas de laisser ouvertes les fenêtres de terminal après le démarrage de Node-RED, aussi bien sous Windows que Linux, et de ne pas les fermer tant que Node-RED doit rester opérationnel !

Installation de Nodes ou de Flows supplémentaires

La carte Arduino n'étant pas disponible par défaut sur Node-RED, il vous faudra l'installer en ajoutant un nouveau *Node* à la palette. Il est possible de procéder de différentes manières. L'installation directe à partir de Node-RED est naturellement un très bon choix. Nous l'avons abordé dans le montage n° 24 sur les langages de programmation par blocs, lorsque nous avons ajouté le Node Arduino. L'autre variante consiste à se rendre sur un site Internet spécifique puis à exécuter manuellement une ou plusieurs commandes dans la ligne de commande. Je vais vous montrer les deux méthodes.

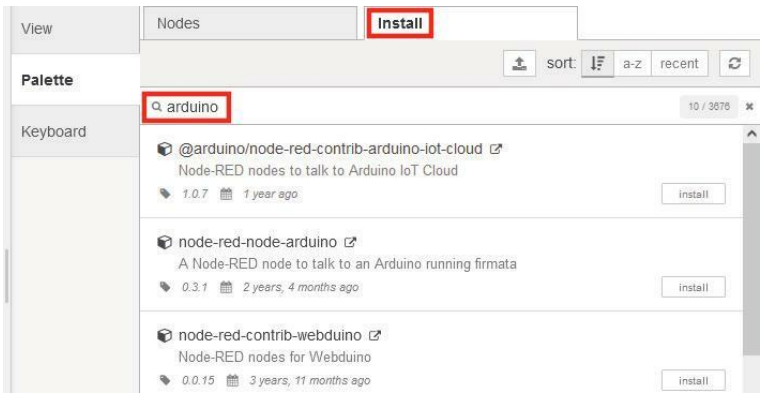
Installation d'extensions à partir de Node-RED

Pour que l'installation de Nodes supplémentaires à partir de Node-RED puisse fonctionner, il faut, bien sûr, avoir démarré Node-RED. Ouvrez Node-RED dans votre navigateur et cliquez sur les trois lignes horizontales en haut à droite pour ouvrir la configuration et sélectionnez l'option de menu *Manage palette* :



◀ Figure 26-8
Gestion de la palette de Node-RED

Une boîte de dialogue s'ouvre. Allez dans l'onglet *Install* et saisissez le terme de votre choix dans la barre de recherche. J'ai ainsi obtenu une liste de résultats après avoir entré le terme *arduino*. Vous pouvez maintenant installer l'extension souhaitée sous la forme d'un Node supplémentaire en cliquant sur le bouton *Install*.



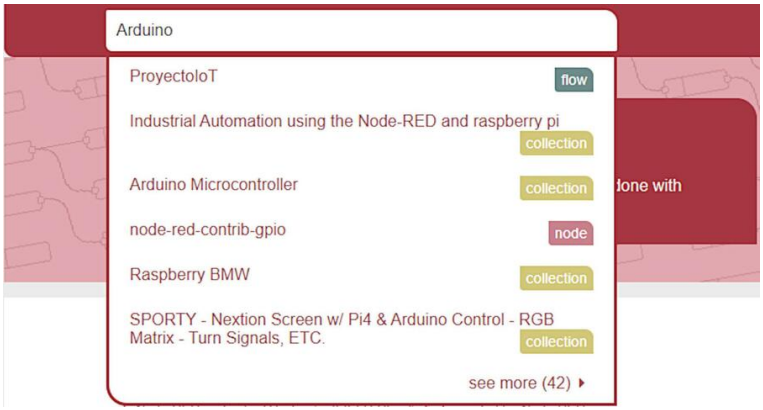
Installation d'extension à l'aide de la ligne de commande

Pour installer manuellement le Node que vous souhaitez, rendez-vous sur la page Internet suivante :

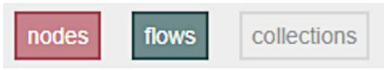
http://flows.nodered.org/

Vous pouvez restreindre l'affichage des Nodes ou des Flows dans la recherche. Entrez simplement le terme *Arduino* dans la barre de recherche. Quelques résultats sont déjà mis en évidence pendant la saisie :

Figure 26-10 ▶
Recherche d'extensions pour Node-RED sur la page Internet

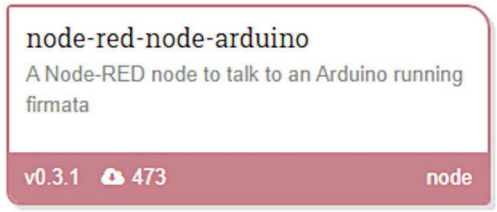


Vous pouvez voir sous la fenêtre de recherche trois boutons qui vous permettent de limiter la recherche aux Nodes, aux Flows ou aux Collections :



L'extension souhaitée s'affiche dans un encadré de couleur rose :

Figure 26-11 ▶
Extensions Arduino pour Node-RED sur la page Internet



Cliquez dessus pour faire apparaître sur une nouvelle page web des informations supplémentaires expliquant comment installer l'extension que vous avez sélectionnée. Pour communiquer avec la carte Arduino, on utilise firmata (déjà vu dans le montage n° 25), qui est un firmware fourni d'office dans chaque environnement de développement Arduino. Une fois l'installation réussie, arrêtez et redémarrez le serveur Node-RED. Avec le Raspberry Pi, cela s'effectue à l'aide des commandes que vous connaissez déjà, dans l'ordre suivant :

```
# node-red-stop
# node-red-start
```

Sur un système Windows, terminez le traitement par lots en appuyant sur Ctrl+C, confirmez avec O et relancez la commande pour démarrer Node-RED :

```
node-red -v
```

Une fois le navigateur reconnecté, qui naturellement perd brièvement la connexion au serveur suite à cette action, vous voyez s'afficher une nouvelle palette portant le nom *Arduino* avec deux Nodes :

Figure 26-12
Nodes Arduino

Maintenant, tout est prêt pour établir la communication entre Node-RED et votre carte Arduino.

Préparation de la carte Arduino Uno

Pour que la communication fonctionne entre votre carte Arduino Uno et votre serveur Node-RED (l'ordinateur local ou le Raspberry Pi), vous devez télécharger sur votre carte Arduino Uno le sketch *StandardFirmata* en tant que firmware à partir du répertoire d'exemples.

Si vous n'avez pas encore installé l'environnement de développement Arduino sur votre Raspberry Pi (nous l'avons fait dans le montage n° 25 sur l'interaction entre Arduino et Raspberry), exécutez les lignes suivantes dans une fenêtre de terminal de votre Raspberry Pi :

```
# sudo apt-get update
# sudo apt-get install arduino
```

Le sketch *StandardFirmata* se trouve dans *Fichiers* | *Exemples* | *Firmata*.

Node-RED dans le navigateur

Avant de commencer, voici un petit aperçu de la façon dont se présente Node-RED dans le navigateur et comment il est organisé. Une fois que vous avez entré dans le navigateur l'adresse IP attribuée, Node-RED s'affiche comme suit. Vous pouvez voir que j'y ai déjà ajouté un Flow simple :

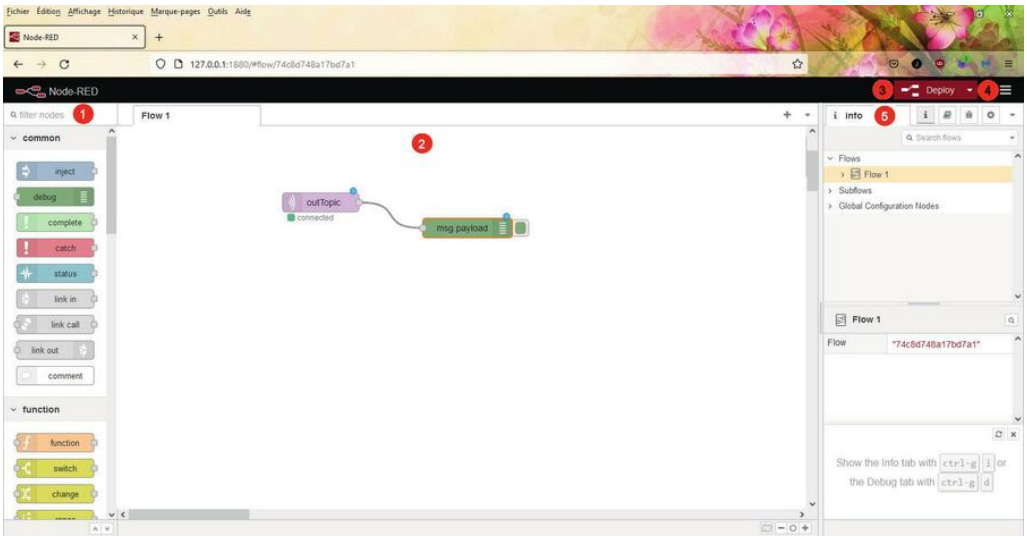


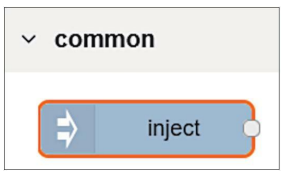
Figure 26-13 ▲ Node-RED dans le navigateur

Les différentes palettes sont représentées dans la partie gauche 1. Pour le moment, c'est la palette *common* qui est ouverte et les Nodes qu'elle contient sont affichés. La partie centrale 2 est un grand espace où peuvent être créés les Flows, organisés en différents onglets. Cliquez sur la case + en haut à droite pour ajouter un nouveau Flow dans la barre d'onglets. Pour activer un Flow, cliquez sur le bouton *Deploy* 3. Pour configurer Node-RED, il vous suffit d'ouvrir un menu 4. Des informations détaillées 5 sur le Node sélectionné s'affichent dans l'onglet *Info*, des informations sur le Flow après son activation dans l'onglet *Debug*.

Il est maintenant temps de démarrer un Flow.

Le Flow du clignotant

Nous commencerons (pourrait-il en être autrement ?) avec un exemple clignotant. Nous allons construire ce Flow pas à pas et je vais vous expliquer tous les détails importants. De quels Nodes avons-nous besoin ? Nous les trouverons dans la partie gauche du Node *inject*, situé dans la palette *common*.



Le Node *inject* vous permet d'injecter un message dans le Flow, manuellement en cliquant sur le bouton *Node* ou à intervalle régulier via le minuteur.

Pour obtenir des informations sur le Node sélectionné, ouvrez le *volet d'informations* en appuyant sur Ctrl+barre d'espace. Cette fenêtre reste ouverte. Si vous cliquez sur un autre Node, vous obtenez immédiatement les informations correspondantes. Le nom et le type de Node *inject* sont indiqués sur la figure ci-après :

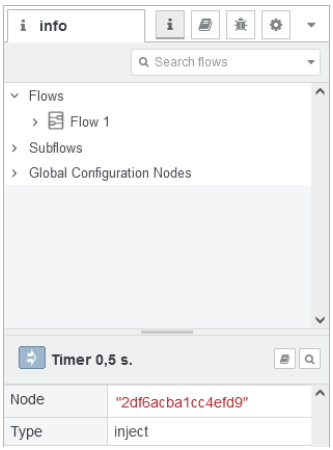


Figure 26-14
Informations sur le Node inject

D'autres indications utiles sur le Node sélectionné s'affichent aussi sous ces informations. Jetez-y un coup d'œil. Cette fenêtre ne nous permet cependant pas de modifier la configuration. Mais alors, comment changer la configuration du Node *inject* ? Double-cliquez sur le Node pour le configurer (cela s'applique aussi à tous les autres Nodes). Une boîte de dialogue s'ouvre et vous pouvez alors modifier les paramètres importants de ce Node.

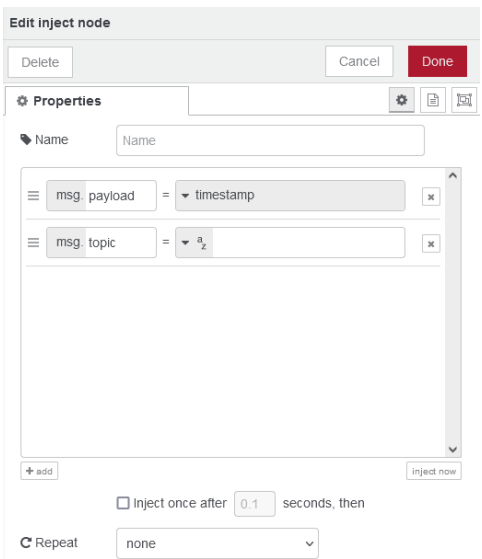


Figure 26-15
Éditer le Node inject

Payload est ce qu'on pourrait appeler la charge utile qui doit être transportée. Dans le cas présent, nous transmettons l'horodatage (*Timestamp*) qui indique à quel moment quelque chose s'est passé. Il s'agit d'une longue succession de chiffres, par exemple :

```
1610897445122
```

Cette succession de chiffres cache la date et l'heure. Vous pouvez ensuite, à l'aide d'un convertisseur d'horodatage, convertir cette valeur dans un format lisible. Il vous suffit de vous rendre sur le site Internet suivant :

https://www.epochconverter.com/

Sur le site, entrez l'horodatage puis cliquez sur le bouton *Timestamp to Human date*, vous obtiendrez alors le résultat suivant :

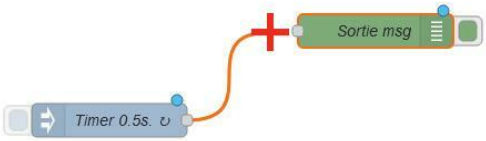
Comment faire pour afficher cet horodatage ? Bon, je vais anticiper un peu. Si vous voulez « mettre au point » un Flow pour le rendre actif, cliquez sur le bouton *Deploy*, qui se trouve en haut à droite de la fenêtre Node-RED. Deploy signifie « Mettre en place » et active ainsi le Flow que vous avez créé sur le serveur.

Si vous voulez envoyer quelque chose une seule fois avec le Node *inject*, ne sélectionnez pas dans la case *Repeat* l'option *interval* mais laisser plutôt l'option *none* :

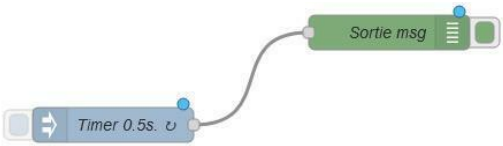
Vous pouvez maintenant entreprendre une action *inject* unique en cliquant sur le petit carré situé à gauche de l'onglet :

Tout est en place. La présence du Node *inject* ne constitue pas pour autant un Flow correct et si vous voulez visualiser la sortie, c'est-à-dire la Payload d'un Node, il faut se servir du Node *debug*. Celui-ci se trouve dans la palette *common* :

Lorsque vous créez le Flow ci-après, quelque chose va se passer dans le volet d'informations de l'onglet *debug*. Les deux Nodes vont se relier aux deux petits points de liaison carrés que vous voyez sur les Nodes. Cliquez sur le petit carré du Node de gauche qui représente la source du Flow, puis tracez un trait de liaison en appuyant sur le bouton gauche de la souris vers le Node de droite qui fonctionne comme la cible du Flow :



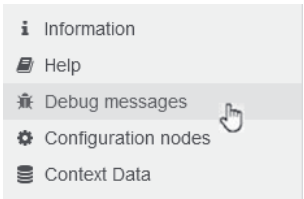
Lorsque vous tracez le trait avec la souris, une ligne de liaison apparaît avec une croix marquant la cible, en rouge ici. Relâchez le bouton de la souris lorsque la croix parvient au-dessus du petit carré du Node cible. Vous venez de créer la liaison, c'est-à-dire le Flow. Il est possible de repositionner les Nodes dans l'espace de travail en maintenant le bouton gauche de la souris appuyé sans supprimer les lignes de liaison. Pour supprimer un Node, sélectionnez-le et appuyez sur la touche Suppr de votre clavier :



Vous devez configurer le Node *inject* pour qu'il puisse envoyer un message toutes les 0,5 s. Pour cela, je règle l'option *Repeat* (répéter) sur *interval* (intervalle) et j'ajoute la valeur 0.5 comme intervalle en secondes. Je nomme le Node *inject Timer 0,5 s* :

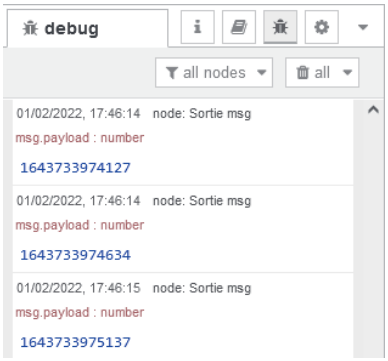
Pour accéder à la fenêtre *debug*, cliquez en haut à droite sur le triangle pointant vers le bas et sélectionnez *Debug messages* :

Figure 26-16 ▶
Fenêtre Debug



Dans mon cas, les messages suivants, dont vous ne voyez ici qu'un petit extrait, ont défilé après le déploiement (*Deploy*) dans la fenêtre *debug* :

Figure 26-17 ▶
Informations Debug



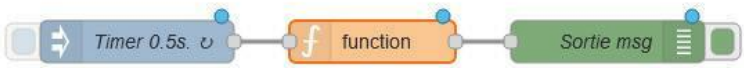
Pour le Flow suivant, vous pouvez encore utiliser le Node *debug*. Il rend bien service lorsque vous recherchez des erreurs. Mais à quoi sert un de ces Nodes *inject* et quelles sont leurs fonctions ? Ce Node sert de minuteur, il envoie toutes les 0,5 secondes un message au Node suivant. L'horodatage ici est en fait secondaire et ne sert qu'à indiquer que quelque chose a été réellement transmis. Comme je l'ai déjà mentionné, nous devons injecter quelque chose à un certain endroit du Flow et c'est ce à quoi va nous servir le Node *inject* ; celui-ci fonctionne quasiment comme un déclencheur et injecte un message dans le Flow à l'intervalle indiqué. Vous vous demandez certainement pourquoi il faut transmettre des informations d'un Node à un autre ou si cela fonctionne en arrière-plan sans que l'on s'en aperçoive. Il s'agit là d'un point essentiel à Node-RED, comparable au paradigme du langage de programmation C++ pour ce qui est de la programmation orientée objets (POO). Node-RED fonctionne de manière comparable. Comme des messages sont échangés entre les Nodes, il existe un objet qui porte le nom *msg* (message). Cet objet possède différentes propriétés (*properties* en anglais). La propriété qui nous importe le plus est Payload, la charge utile qui est transportée. Vous trouverez des informations détaillées sur les propriétés possibles à l'adresse Internet suivante :

https://github.com/node-red/node-red/wiki/Node-msg-Conventions

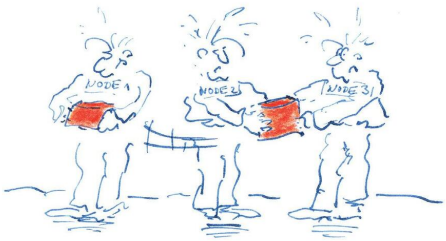
Pour accéder à la Payload, utilisez l'écriture suivante :

```
msg.payload
```

Nous allons voir tout cela dans l'exemple qui suit. Notre objectif est de faire clignoter sur la carte Arduino une certaine broche, de préférence la broche 13 (toujours elle). Pour cela, nous devons transmettre alternativement les valeurs *false* et *true* :



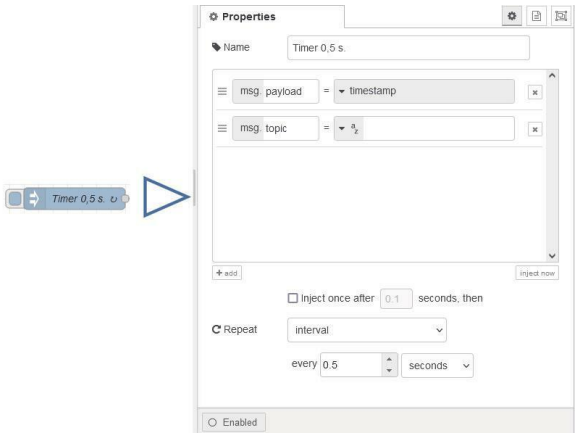
Vous pouvez voir sur la figure ci-après comment les Nodes fonctionnent entre eux et comment ils envoient leurs messages au Node suivant :



Regardons cela de plus près. Je double-clique sur le Node pour afficher ses propriétés.

Le Node *inject*

Le Node *inject* nous permet de générer à l'aide des paramètres indiqués un minuteur, qui envoie un message sous la forme d'un horodatage toutes les 0,5 secondes :



Le Node Fonction

Le Node Fonction qui se trouve dans la palette *function* permet de créer une fonction que nous allons doter d'un code pour clarifier ce qui doit se passer lorsque la fonction est activée. Le code doit être ajouté dans l'onglet *On Message* :

Le contenu, également appelé *body*, nécessite une explication. Chaque Node possède un objet prédéfini qui lui est propre et qui porte le nom `context`. Nous pouvons maintenant insérer des propriétés (*properties*) à l'aide de l'opérateur point (`.`). J'ai ajouté pour le niveau broche de la carte Arduino la propriété `level`. Vous trouverez des informations détaillées sur l'objet `context` à l'adresse Internet suivante :

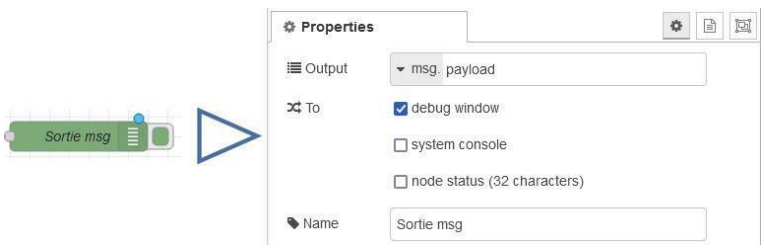
https://nodered.org/docs/creating-nodes/context

À la ligne 1 de la fonction, je nie la valeur à chaque ouverture de celle-ci en utilisant l'opérateur NOT, accessible par le point d'exclamation (`!`). Si la propriété n'est pas encore initialisée au départ, ce qui devrait être le cas, `|| false` entreprend une initialisation.

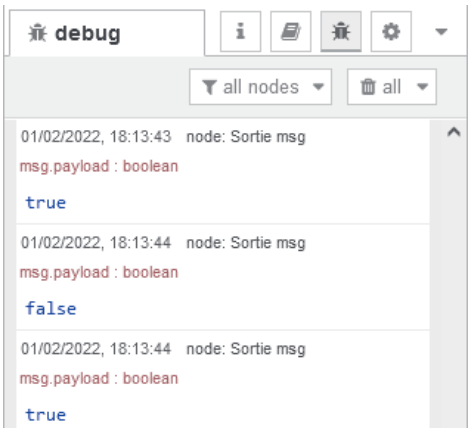
À la ligne 2, le `context.level` de la propriété `payload` est affecté à l'objet `msg` et défini à la ligne 3 comme valeur de retour. Le Node suivant reçoit donc l'objet `msg` complet comme entrée. Dans notre cas, il s'agit du Node *debug*.

Le Node debug

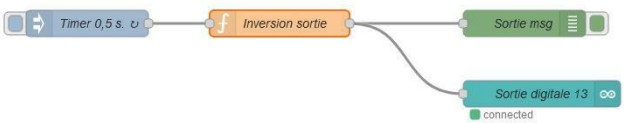
Le Node *debug* va nous permettre d'indiquer la sortie (l'édition) de la fonction :



Dans mon exemple, vous voyez que l'édition (Output) est la propriété Payload, qui est éditée dans l'onglet *debug*. Regardons de plus près cette édition en allant faire un petit tour sur l'onglet *debug* correspondant :



Vous voyez que la propriété `payload` alterne entre les valeurs `false` et `true`. Parfait ! Vous pouvez maintenant attacher le Node *Arduino* au Node *function* sans que le Node *debug* ne gêne ni qu'il disparaisse. Votre Flow prend l'apparence suivante et se divise après le Node *function*. L'édition (Output) est dirigée vers les deux Nodes ajoutés (Payload et Arduino) :



La configuration du Node Arduino (Arduino-Out) qui se cache dans la palette *Arduino* que nous avons ajoutée se présente comme suit :

Les ports COM sont disponibles sur un ordinateur Windows. En revanche, sous Linux avec un Raspberry Pi, le port est */dev/ttyACM0*. Dans *Type*, sélectionnez la commande des broches numériques et indiquez ensuite le bon numéro de broche. Le champ *Name* peut rester vide, vous pouvez aussi donner un autre nom si Pin 13 ne vous plaît pas. Une fois que vous avez cliqué sur le bouton *Deploy*, la LED Onboard (Broche 13) doit clignoter sur votre carte Arduino Uno. Vous trouverez des informations détaillées sur les fonctions Node-RED à l'adresse Internet suivante :

https://nodered.org/docs/writing-functions

La communication entre votre carte Arduino et le serveur Node-RED s'opérait jusqu'ici par le biais du sketch firmata. Peut-on faire autrement ?

Montage avec le capteur de température et d'humidité DHT11

Il existe un capteur très intéressant, qui peut mesurer aussi bien la température que l'humidité. Il s'agit du capteur DHT11 (vu brièvement dans la fin du montage n° 22) et celui-ci est tout à fait approprié pour faire la démonstration de la communication entre la carte Arduino et Node-RED. Mais vous pouvez aussi utiliser n'importe quel autre capteur que vous aurez raccordé à votre carte Arduino.

Figure 26-18 ▶
Capteur de température et d'humidité DHT11

Il ne présente que trois ports et le mieux est de le connecter à la carte Arduino via une petite carte porteuse, sur laquelle se trouve déjà la résistance supplémentaire dont vous avez besoin. Rappel des caractéristiques du DHT11 :

- tension d'alimentation 3,3 V à 5 V ;
- plage de mesure de l'humidité relative de l'air 20 % à 95 % ;
- tolérance de la plage de mesure pour l'humidité relative de l'air ±5 % ;
- plage de mesure de la température 0 °C à 60 °C ;
- tolérance de la plage de mesure pour la température ±2 %.

Composants nécessaires

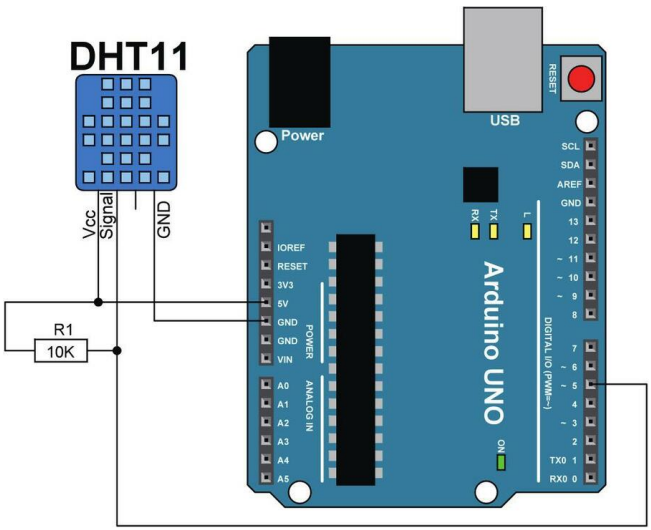
Pour ce montage, nous avons besoin des composants suivants :

Composant	
1 capteur DHT11	
1 résistance de 10 kΩ (si elle n'est pas déjà sur la carte porteuse)	

◀ **Tableau 26-1**
Liste des composants

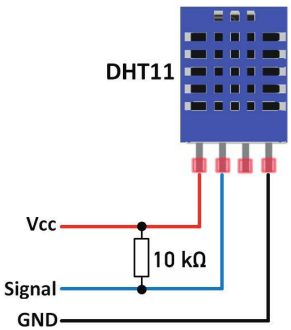
Schéma

Le schéma des connexions est simple et consiste uniquement à alimenter le capteur en tension et à raccorder la broche véhiculant les signaux.



◀ **Figure 26-19**
Schéma des connexions pour l'interrogation du DHT11

Si le capteur DTH11 n'est pas sur une carte porteuse, vous devrez alors le raccorder comme suit :



◀ **Figure 26-20**
Raccordement externe du DHT11

JAVASCRIPT OBJECT NOTATION-FORMAT (JSON)

Je voudrais vous faire découvrir un format de données appelé JSON. Il s'agit de l'abréviation de JavaScript Object Notation. Ce format est très compact et facile à lire pour les humains, il a été développé pour échanger des données entre différentes applications. Il est maintenant largement utilisé dans les applications web et s'est également établi depuis longtemps dans le monde des développeurs amateurs. Ce format de données ne devrait donc pas avoir de secrets pour vous. Vous trouverez des informations détaillées sur ce thème à l'adresse Internet suivante :

https://fr.wikipedia.org/wiki/JavaScript_Object_Notation

Je vais vous montrer comment se présente ce format à l'aide d'un exemple concret que nous allons aussi mettre en pratique pour ce montage. La structure suivante nous montre un format JSON typique :

```
{
"Type"        : "11",
"Status"      : "OK",
"Humidity"    : 13.0,
"Temperature" : 20.0
}
```

On trouve toujours une paire nom/valeur reliés par deux points. Le nom est toujours une chaîne de caractères, par exemple Type, et la valeur peut être :

- un objet ;
- un tableau ;
- une chaîne de caractères ;
- un nombre ;
- *true*, *false* ou *null*.

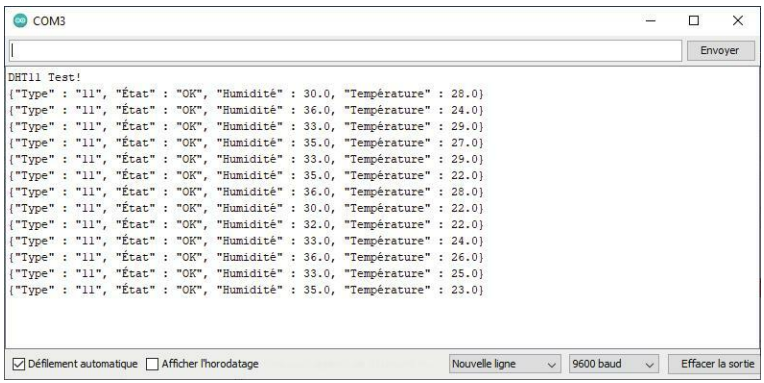
Les paires nom/valeur sont séparées les unes des autres par des virgules et sont encapsulées par des accolades. Si nous programmons le sketch Arduino de sorte qu'il fournisse les valeurs à envoyer sous cette forme vers une interface sérielle, nous pourrons alors évaluer le flux de données dans Node-RED. Notre objectif est de procéder ensuite à une évaluation et de générer des messages adéquats par e-mails et/ou via Twitter.

Sketch Arduino

Voici à quoi ressemble le sketch Arduino. Vous devez préalablement avoir installé la bibliothèque DHT11. Pour commencer, je vais vous montrer encore une fois le sketch Arduino servant à afficher les valeurs mesurées de la température et de l'humidité dans le moniteur série :

```
#include <DHT.h>          // Inclure la bibliothèque
#define DHTPIN 8          // Broche numérique
#define DHTTYPE DHT11     // DHT11
DHT dht(DHTPIN, DHTTYPE); // Initialisation de l'objet

void setup() {
  Serial.begin(9600);
  Serial.println("DHT11 Test!");
  dht.begin();
  randomSeed(analogRead(0));
}

void loop() {
  // Mesurer l'humidité
  float h= dht.readHumidity();
  // Mesurer la température en degrés Celsius
  float t= dht.readTemperature();t=11.0;
  h=random(30.0,40.0);
  t=random(20.0,30.0);
  // Vérifier les mesures
  if(isnan(h) || isnan(t)) {
    Serial.println(«Erreur lors de la lecture du capteur DHT!»);
    return;  // Exit
  }
  Serial.print("{");
  Serial.print("\"Type\" : \"");
  Serial.print(DHTTYPE);
  Serial.print(«\», «);
  Serial.print(«\»État\» : \»»);
  Serial.print("OK");
  Serial.print("\", ");
  Serial.print("\"Humidité\" : ");
  Serial.print(h, 1);
  Serial.print(«, «);
  Serial.print(«\»Température\» : «);
  Serial.print(t, 1); Serial.println(„}");
  delay(60000); // Pause de 1 minute
```

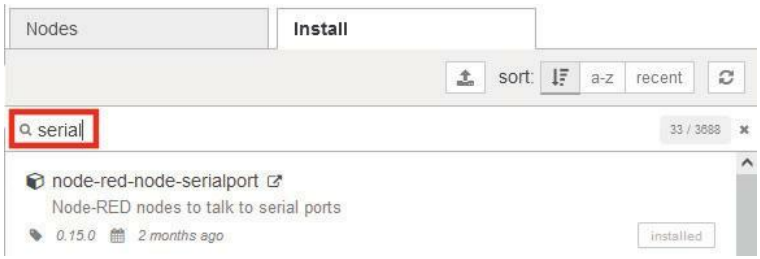
L'édition sur le moniteur série se présente comme suit :



◀ **Figure 26-21**
Édition des valeurs de mesure au format JSON du capteur DHT11 sur le moniteur série

Vous voyez ici que cela n'a rien à voir avec le sketch firmata mais qu'il s'agit d'un sketch visant à interroger le capteur DHT11, qui envoie les valeurs mesurées vers une interface sérielle. Est-il possible avec Node-RED d'interroger directement l'interface sérielle sans passer par le Node Arduino installé au préalable ? Bien sûr que oui. Si le Node *Serial* n'est pas encore installé sur votre serveur Node-RED, vous pourrez le faire ultérieurement. Je l'ai fait dans Node-RED à l'aide de la gestion de la palette en recherchant le terme *serial* puis en cliquant sur *Install*.

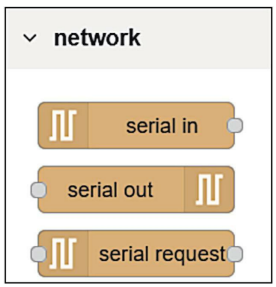
Figure 26-22 ▶
Installation du Node Serial



ATTENTION AU PORT COM

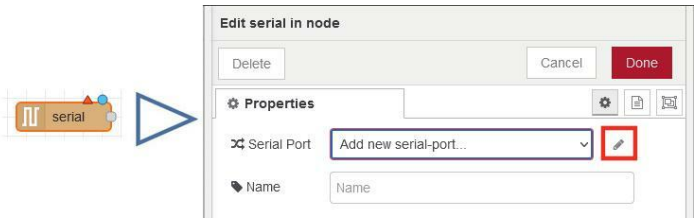
La carte Arduino et Node-RED utilisent le même port COM. Il est donc important de ne plus bloquer ce port COM sur votre carte Arduino Uno par l'environnement de développement Arduino une fois que le sketch a été chargé. Dans le sens inverse, Node-RED bloque le port COM en cas d'accès à celui-ci et exclut l'environnement de développement Arduino. N'oubliez pas ce facteur très important ! En cas d'urgence, débranchez la carte Arduino de l'ordinateur puis rebranchez-la ensuite.

Nous recommençons depuis le début en créant le Flow ci-après. Dans la palette *network*, vous trouverez après l'installation trois nouveaux Nodes, parmi lesquels le Node *serial in* dont vous allez avoir besoin.

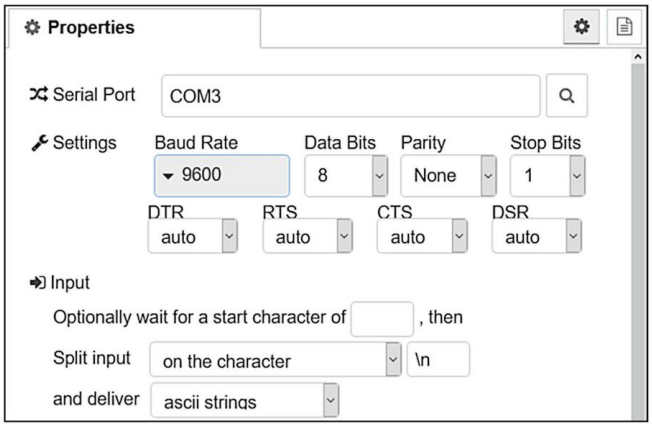


Déplacez ce Node dans le Flow et ajoutez un Node *Debug* pour afficher le flux de données sériel. Vous obtenez :

Vous devez configurer le Node *serial* avec les bons paramètres, qui se présentent chez moi comme indiqués ci-après. Le débit en bauds doit correspondre à celui de la carte Arduino. Après avoir double-cliqué sur le Node *serial in*, cliquez sur le symbole du crayon encadré en rouge, car il n'y a pas encore de port COM sélectionné pour la configuration :

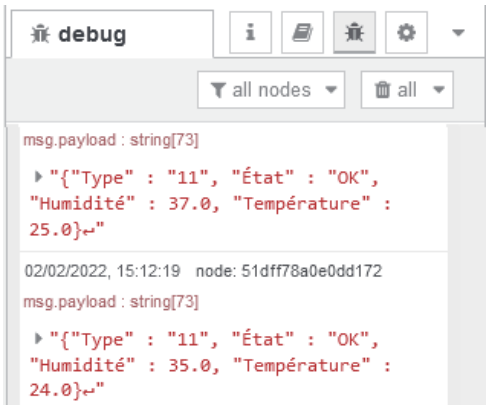


Les paramètres de la connexion devront ensuite être réglés de façon à ce que la communication avec votre carte Arduino s'opère sans encombre :



Après un déploiement (Deploy), les messages suivants défilent dans la fenêtre *debug* :

Figure 26-23 ▶ Édition des valeurs mesurées par le capteur DHT11 dans la fenêtre debug

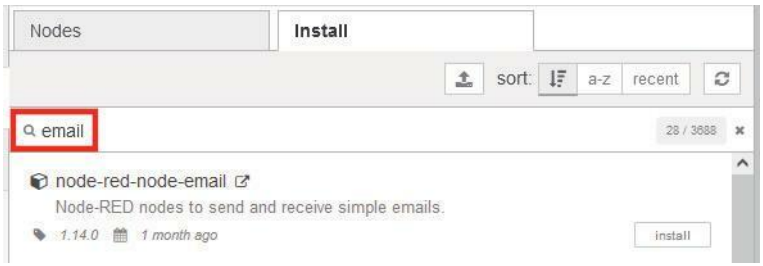


Cette édition est la même que celle qui s'affiche dans le moniteur série. Les valeurs sont des valeurs nues, sans aucune évaluation. Il nous reste à procéder à celle-ci. Nous modifions le Flow de façon à soumettre les données brutes à une évaluation et à générer des messages correspondants. Ces messages devront ensuite être envoyés par e-mail.

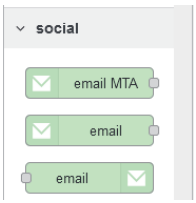
Envoi par e-mail

Si le Node E-Mail n'est pas encore installé sur votre serveur Node-RED, vous pourrez l'installer plus tard. Je l'ai fait dans Node-RED à l'aide de la gestion de la palette en recherchant le terme *email* puis en cliquant sur *Install* :

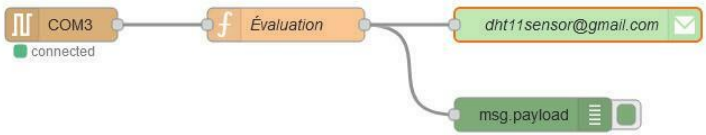
Figure 26-24 ▶ Installation du Node E-Mail



Après l'installation, trois nouveaux Nodes apparaissent dans la palette sous *social* :



Vous pouvez maintenant structurer le Flow ci-après en suivant mes explications pas à pas. Vous pouvez voir que le flux de données se divise après le Node Évaluation et que les deux Nodes suivants reçoivent les mêmes informations :



Passons en revue les différents Nodes. Vous connaissez déjà le Node Serial et vous voyez grâce au voyant vert avec la mention *connected* qu'une connexion sérielle est établie entre Node-RED et votre carte Arduino Uno. L'évaluation complète s'effectue dans le Node Fonction qui porte ici le nom *Évaluation*.

Examinons de plus près le code, car il est un peu complexe. On crée au début une nouvelle variable avec le nom `var` qui vérifie et apparie la Payload à l'aide de la classe JSON et de la méthode `parse`. Un objet vide `msg.env` est ensuite créé, servant à recueillir les informations qui seront fournies par la carte Arduino via l'interface sérielle, en l'occurrence le type de capteur, l'état, l'humidité et la température.

Aux lignes suivantes, ces valeurs sont affectées à l'objet `msg.env`, dans lequel les propriétés du format JSON sont appliquées.

```
// Analyse par JSON
var raw = JSON.parse(msg.payload);

msg.env = {}; // Nouvel objet
var returnMsg = ""; // Message de retour
// Affectation des valeurs JSON
msg.env.Type = raw.Type; // Type de capteur
msg.env.Status = raw.État; // État du capteur
msg.env.Humidity = raw.Humidité; // Humidité du cpateur
msg.env.Temperature = raw.Température; // Température du capteur
```

La figure ci-après montre sur le côté droit les données quasi brutes qui entrent dans Node-RED sous la forme d'une chaîne de caractères JSON :

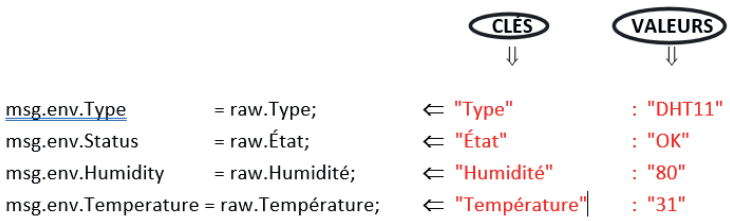


Figure 26-25
Affectation des données brutes à partir du format JSON à l'objet msg.env

Ces données brutes enregistrées dans l'objet `raw` peuvent être adressées à partir du flux de données JSON via les noms enregistrés. Elles sont affectées à l'objet `msg.env` avec les noms en anglais que j'ai conservés lors de l'affectation. La variable `returnMsg` va alors recueillir aussitôt le message qui devra être envoyé à l'adresse mail. J'ai cependant programmé une petite évaluation qui s'effectue via les instructions `if` de sorte qu'un message soit ajouté lorsque les valeurs seuils que j'ai définies sont dépassées, en l'occurrence lorsque la température ou l'humidité est trop élevée. Le code suivant doit être également ajouté pour compléter le Node Évaluation :

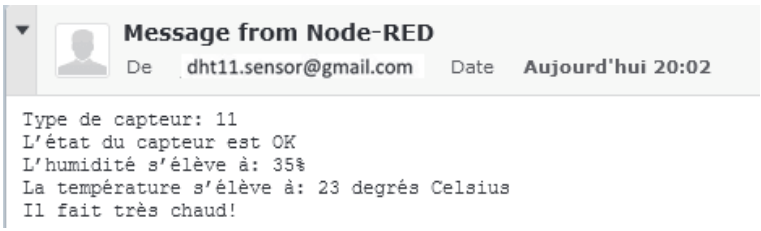
```
// Type de capteur
returnMsg = "Type de capteur: " + msg.env.Type + "\n";

// Evaluation de l'état
if (msg.env.Status === "OK")
returnMsg += "L'état du capteur est OK\n";
// Type de capteur
returnMsg = "Type de capteur: " + msg.env.Type + "\n";

// Evaluation de l'état
if (msg.env.Status === "OK")
    returnMsg += "L'état du capteur est OK\n";
else
    returnMsg += "L'état du capteur n'est pas OK\n";

// Evaluation de l'humidité
returnMsg += "L'humidité s'élève à: " + msg.env.Humidity + "%\n";
if (msg.env.Humidity > 50)
    returnMsg += "Il fait très humide!\n";

// Evaluation de la température
returnMsg += "La température s'élève à: " + msg.env.Temperature + " degrés Celsius\n";
if (msg.env.Temperature > 24)
    returnMsg += "Il fait très chaud!\n";

msg = {payload : returnMsg};
return msg;
```

La syntaxe utilisée se base sur le langage de programmation JavaScript. Il existe de nombreux tutoriels sur Internet qui proposent une introduction facile à ce langage de programmation. Enfin et surtout, nous envoyons le message créé à l'aide du Node E-Mail de la palette *social*. La figure ci-après montre la configuration dans laquelle j'ai utilisé un compte e-mail test. Inscrivez ici les données de connexion du compte personnel que vous souhaitez utiliser pour envoyer les valeurs de mesure :

Lorsque vous chargez le Flow sur le serveur, les valeurs du capteur DHT11 seront envoyées par e-mail à l'intervalle que vous avez défini dans le sketch

Arduino. Pour le moment, une pause de 60 secondes est fixée entre les envois. Vous pouvez la modifier ou bien envoyer un message dans Node-RED seulement lorsque certaines valeurs seuils sont dépassées ou ne sont pas atteintes. Je vois alors dans ma boîte de messagerie électronique le message suivant qui a été envoyé.

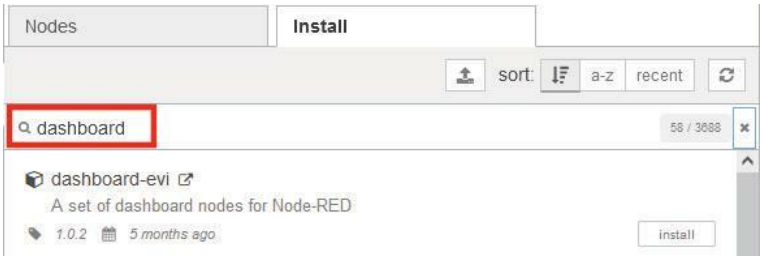
Cet e-mail affiche le contenu suivant :



Vous souhaitez peut-être afficher ces informations sous une autre forme que de simples valeurs numériques, avec une présentation graphique par exemple. Dans ce cas, Node-RED est un outil aussi simple que formidable.

Le Dashboard

Le Dashboard, ou tableau de bord en français, offre une grande variété de graphiques sous la forme par exemple de tachymètres, de diagrammes à poutres ou de camemberts. Pour les utiliser dans Node-RED, vous devez ajouter les Nodes *Dashboard* dans la palette :

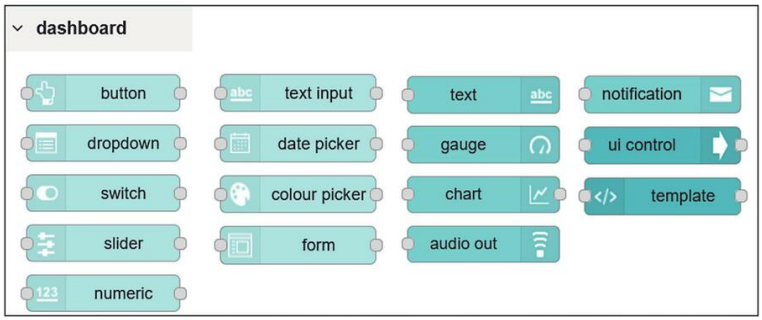


◀ **Figure 26-26**
Installation du Node Dashboard

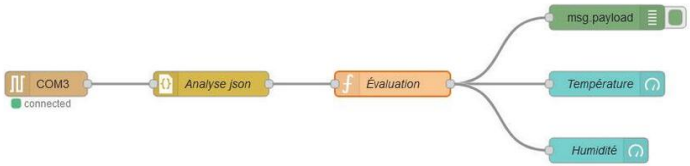
ENVOI D'E-MAILS À PARTIR D'APPLICATIONS

Presque tous les fournisseurs de messagerie électronique ont mis en place des mesures de sécurité qui ne permettent pas facilement d'envoyer des e-mails à partir de programmes, les fameuses applications. Cette fonctionnalité doit être débloquée dans les paramètres du compte de messagerie électronique et/ou un mot de passe spécifique doit être créé pour autoriser cette fonctionnalité. L'envoi d'e-mails ne doit pas non plus s'effectuer dans un intervalle trop rapproché sinon une attaque de spam est suspectée, pouvant être à l'origine de problèmes.

Après l'installation, de très nombreux Nodes apparaissent dans la palette sous *Dashboard* :

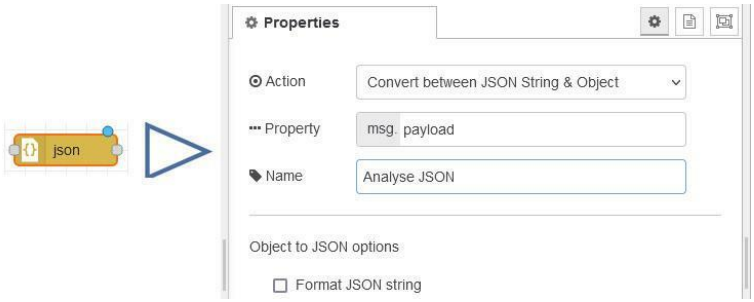


Avant de commencer, vous devez adapter le sketch Arduino de façon à régler une pause d'une seconde environ (au lieu des 60 secondes) entre l'envoi des valeurs mesurées. Les valeurs mesurées seront ainsi représentées presque en temps réel ! Vous pouvez maintenant structurer le Flow suivant en reprenant mes explications pas à pas :



Immédiatement après le Node COM de la carte Arduino se trouve le Node *json*, prélevé de la palette dans la catégorie *parser* :

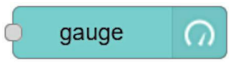
Procédez à la configuration de ce Node comme indiqué ci-après. Notez que la conversion d'une chaîne de caractères en un objet JavaScript et vice versa est :



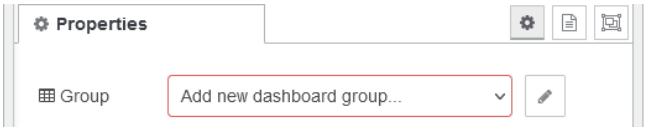
Le Node *function* ci-après a pour tâche d'extraire les valeurs mesurées de l'objet JavaScript. Le code se présente comme suit :

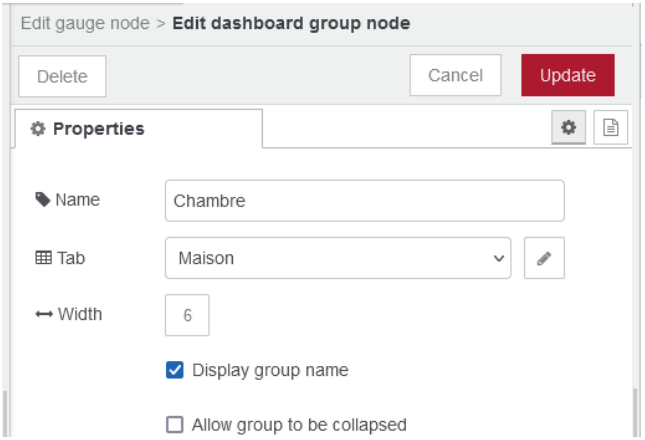
```
msg.env = {};
msg.env.Humidity    = msg.payload.Humidité;
msg.env.Temperature = msg.payload.Température;
return msg;
```

Les informations des deux valeurs mesurées pour la température et l'humidité sont donc enregistrées dans `msg.Temperature` et `msg.Humidity` et seront aussitôt utilisées pour l'affichage dans les graphiques correspondants. Nous utiliserons à cet effet le Node *gauge* dans le tableau de bord que nous venons d'installer.

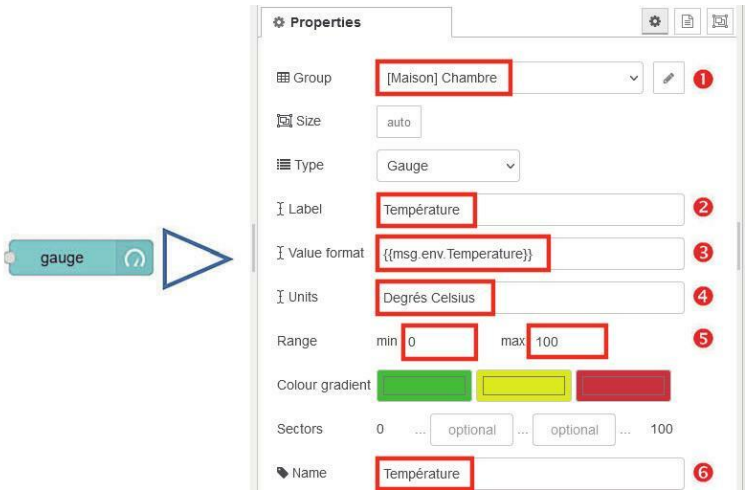


En premier lieu, il vous faudra affecter ce Node à un groupe UI que vous devez d'abord créer. UI est l'abréviation de *User Interface* (interface utilisateur en français) et recouvre la présentation que nous allons obtenir dans le navigateur. Double-cliquez sur le Node, cliquez sur le crayon et entrez un nom pour le décrire :





La configuration du Node Gauge est un peu complexe et se présente comme suit. J'ai numéroté sur la figure les paramètres importants :

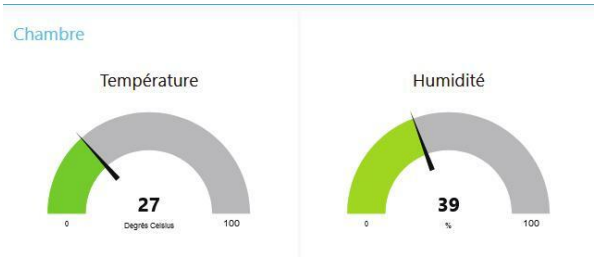


1. Affectation du groupe UI qui vient d'être créé
2. Inscription de l'affichage
3. Valeur à afficher, ici *msg.env.Temperature*
4. Indication de l'unité de mesure devant s'afficher sous la valeur de mesure
5. Plage à l'intérieur de laquelle la valeur de mesure peut varier
6. Nom de la mesure

Mais comment voir l'interface utilisateur après un déploiement ? Ce n'est pas possible à ce stade. Vous avez besoin pour cela d'une extension de votre URL, qui est, pour l'ordinateur local :

```
http://127.0.0.1:1880/ui/.
```

Il vous suffit par conséquent d'ajouter `/ui/` à la fin de l'adresse que vous connaissez. Le mieux est d'ouvrir une autre fenêtre de navigateur pour ne pas fermer l'environnement de développement Node-RED. Dans l'exemple indiqué, il reste à afficher *Humidity*, l'humidité, par le biais du Node *Gauge* en dessous. Je vous laisse faire tout seul. Pour ma part, j'obtiens par exemple le résultat suivant après avoir soufflé sur le capteur DHT11 :



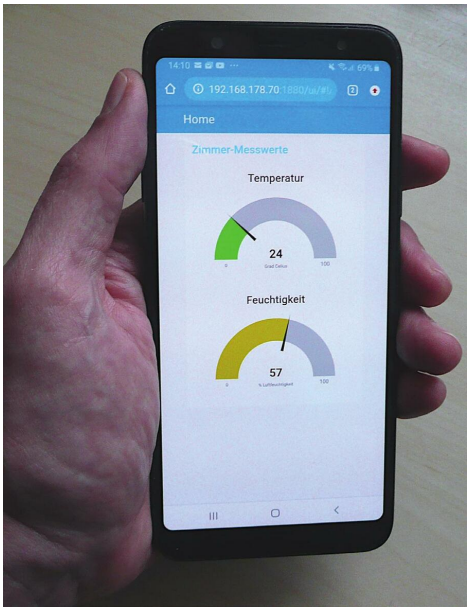
◀ **Figure 26-27**
Valeurs de mesure dans l'UI de Node-RED

Affichage des valeurs de mesure sur un smartphone

Vous pouvez aussi afficher l'interface utilisateur sur un smartphone ou une tablette en indiquant l'adresse IP (et non l'ordinateur local !). Dans mon cas, l'URL à entrer est celle-ci :

```
http://192.168.178.70:1880/ui/
```

Mon smartphone montre immédiatement la page suivante :

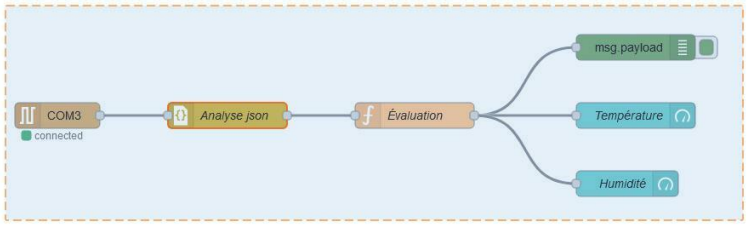


◀ **Figure 26-28**
Valeurs de mesure sur un smartphone

Exporter le code à partir de Node-RED

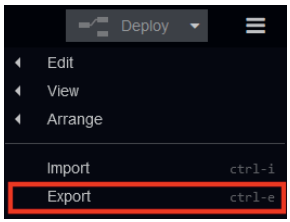
Lorsque vous créez un Flow, celui-ci est enregistré au format JSON indiqué. Vous pouvez donc exporter votre Flow et le mettre le cas échéant à la disposition d'autres utilisateurs. Pour cela, encadrez tous les Nodes en maintenant le bouton gauche de la souris enfoncé :

Figure 26-29 ▶
Nodes sélectionnés

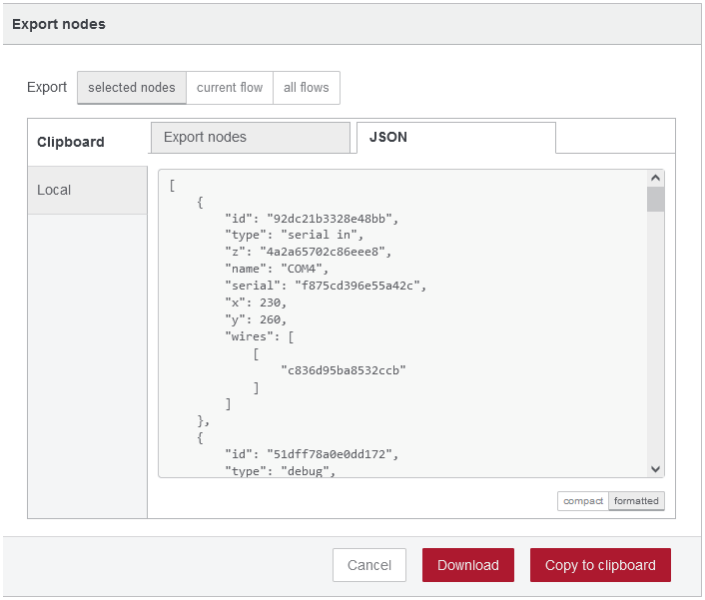


Allez ensuite dans le menu en cliquant sur les trois traits horizontaux blancs près du bouton *Deploy* en haut à droite et sélectionnez *Export* :

Figure 26-30 ▶
Préparer l'exportation

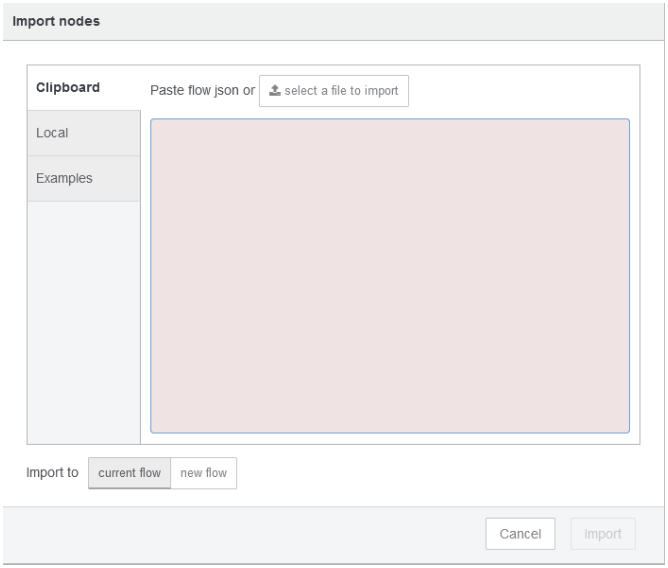


Vous voyez alors la fenêtre ci-après avec le code JSON correspondant. Vous pouvez choisir de télécharger le code ou de le copier dans le presse-papier :



◀ **Figure 26-31**
Télécharger les Nodes ou les exporter dans le presse-papier

Vous pouvez lire dans Node-RED un code téléchargé via l'option de menu *Import*, soit en le copiant à partir du presse-papier, soit en sélectionnant un fichier correspondant :



◀ **Figure 26-32**
Importation de Nodes dans Node-RED

Cliquez enfin sur le bouton *Import*. Les Nodes s'affichent et peuvent être positionnés à l'aide de la souris.

Problèmes courants

Si la connexion entre Node-RED et la carte Arduino ne fonctionne pas ou si elle est interrompue après un déploiement, débranchez la carte Arduino de l'ordinateur et rebranchez-la. Stoppez l'environnement de développement Arduino, notamment le moniteur série. Si le problème persiste, supprimez le port série dans le menu *Configuration nodes* de Node-RED.

Qu'avez-vous appris ?

- Vous avez découvert les bases de Node-RED et réalisé une installation pour que Node-RED fasse office de serveur. C'est possible sur un Raspberry Pi, un ordinateur sous Windows ou un Mac.
- Vous avez créé et affiché des messages.
- Vous avez raccordé le capteur DHT11 à votre carte Arduino Uno puis envoyé des messages au serveur Node-RED via l'interface sérielle. Ces valeurs ont ensuite été transmises par e-mail.
- Vous avez visualisé ces valeurs de mesure à l'aide d'un tableau de bord via une page web, que vous avez pu ouvrir sur un smartphone ou une tablette.

Montage 27

MQTT

Je vais utiliser dans ce montage le Raspberry Pi comme serveur pour un logiciel nommé MQTT. Ce dernier est un protocole de messagerie client-serveur. Après l'établissement d'une connexion, des clients envoient des messages au serveur, un Raspberry dans le cas présent. Le serveur MQTT collecte et prépare les messages reçus sous la forme d'une base de données. L'architecture client-serveur de cette application apporte de la clarté et de l'ordre dans vos projets. C'est pourquoi nous allons aborder ce thème dans ce montage.

Communication M2M avec MQTT

MQTT est l'abréviation de *Message Queue Telemetry Transport*. Ce protocole de messagerie (*Publishing* & *Subscribe*) a été créé pour la communication de machine à machine (M2M). Il constitue un précieux outil lorsqu'il s'agit d'envoyer des informations dans des réseaux avec une bande passante limitée et une latence (temps de réaction) élevée pour commander par exemple des actionneurs ou extraire des données par l'intermédiaire de capteurs. Il a été développé en 1999 par la société IBM pour la communication par satellite et s'est standardisé depuis 2013 comme protocole pour l'Internet des objets, l'IoT. Les ports 1883 et 8883 sont officiellement réservés à la transmission au sein d'un réseau. Le site Internet officiel est disponible à l'adresse suivante :

https://mqtt.org/

Dans les architectures client-serveur, qui sont aujourd'hui un composant essentiel de presque tous les réseaux, des données sont échangées entre les deux instances. Pour ce montage, je vais utiliser le Raspberry Pi comme serveur MQTT, bien qu'il existe aussi des installations pour tous les systèmes d'exploitation usuels comme Windows ou macOS. Mais me direz-vous, MQTT, c'est quoi exactement et à quoi ça sert ?

De plus en plus d'appareils sont qualifiés de *smart*. Cela peut être des réfrigérateurs, des lave-linges, des grille-pains ou des appareils plus banals comme des brosses à dents électriques ou des montres-bracelets. S'il est possible d'introduire des circuits électroniques dans l'appareil, celui-ci devient alors un *Smart Device*, capable de recevoir et d'envoyer des données. D'autres domaines d'application plus indispensables ont fait leur apparition, comme les appareils implantés qui mesurent le taux de glycémie ou les stimulateurs cardiaques qui transmettent des données biologiques.

C'est justement en cas de liaisons sur de grandes distances dans des réseaux instables que le protocole MQTT léger s'est établi comme protocole standard. MQTT est fondé sur le TCP (*Transmission Control Protocol*) comme protocole de transport, garantissant la transmission des messages lorsque la connexion est établie.

Conventions de nommage dans MQTT

Avec MQTT, l'ensemble fonctionne de manière similaire, avec toutefois une différence pour ce qui est de la dénomination. Mais est-ce vraiment un problème pour nous dans la mesure où le résultat est le même ? Nous avons un expéditeur, le *Publisher* (l'éditeur) et un destinataire, le *Subscriber* (l'abonné). Entre ces deux instances intervient un serveur, qui fait office d'intermédiaire, il est appelé *Broker*. J'ai résumé sur la figure ci-après ces relations pour les rendre plus claires :

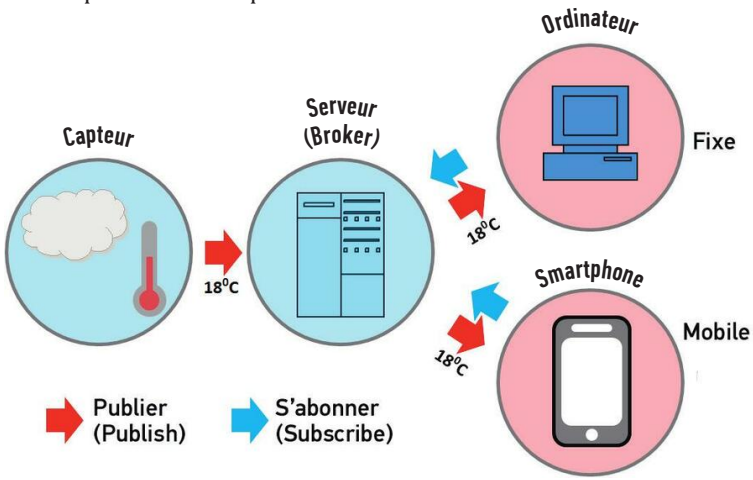


Figure 27-1 ▶
L'architecture MQTT de base

Le capteur de température représenté ici envoie (ou publie) sa valeur de mesure au Serveur (Broker), qui la reçoit et l'enregistre. Le Broker envoie à son tour cette valeur de mesure à d'autres appareils, aussi appelés nœuds, par exemple à un ordinateur fixe ou un terminal mobile (un smartphone) qui travaille avec les informations reçues et qui les évalue. Ces terminaux se sont abonnés aux données.

Structure de messagerie MQTT

Pour garantir un certain ordre au sein de toute cette communication, l'échange de messages s'opère par le biais de *Topics* sous la forme d'une chaîne de caractères. La structure utilisée rappelle une URL servant à ouvrir une adresse Internet sur un navigateur web. Un Topic sert à filtrer les différents messages sur le Broker MQTT. Un Topic peut être par exemple :

Figure 27-2
Topic possible

Vous trouverez ci-après une liste d'exemples pratiques de Topics relatifs à la présence de capteurs dans une maison :

- maison/cave/chaufferie/température ;
- maison/rez-de-chaussée/salon/lumière ;
- maison/rez-de-chaussée/cuisine/réfrigérateur/température ;
- maison/grenier/interrupteur de lucarne de toit ;
- garage/lumière.

Chacun des Topics utilisés doit se composer d'au moins un caractère, les espaces vides étant autorisées, de même que les majuscules et les minuscules.

Métacaractères MQTT

Il est possible d'utiliser des métacaractères pour interroger simultanément plusieurs capteurs. On distingue métacaractères simple et multiple.

Métacaractère Single Level (joker) : +

Comme son nom l'indique, ce métacaractère sous la forme du signe plus (+) renvoie à un seul niveau *(Single Level)*.

Figure 27-3 ▶
Le métacaractère Single Level

Voici quelques exemples expliquant quels sont les effets du métacaractère Single Level utilisé précédemment :

Figure 27-4 ▶
Le métacaractère Single Level et ses effets

Métacaractère Multi Level :

Le métacaractère ci-après sous la forme d'un dièse (#) est un métacaractère Multi Level qui filtre plusieurs niveaux. Il figure toujours à la fin d'un Topic :

Figure 27-5 ▶
Le métacaractère Multi Level

Voici quelques exemples expliquant quels sont les effets du métacaractère Multi Level utilisé précédemment :

Grâce à un filtrage intelligent et des structures logiques, vous pouvez interroger (ou vous abonner à) certains nœuds de façon très simple.

Installation de MQTT

L'installation de MQTT peut s'effectuer sur les systèmes d'exploitation connus. Dans ce montage, je décris l'installation sur un Raspberry Pi. Si vous souhaitez cependant installer le Broker MQTT sur votre système Windows, rendez-vous sur le site Internet suivant :

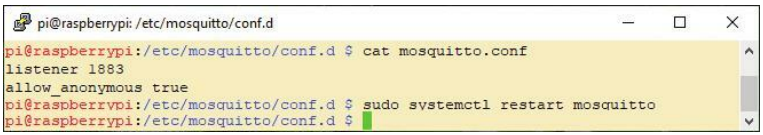
https://mosquitto.org/download/

Vous y trouverez sous la rubrique Windows deux installations binaires, permettant l'utilisation pour 32 bits ou 64 bits. Le tout s'installe très vite et vous pourrez ensuite entreprendre sous Windows avec la ligne d'instructions les exemples MQTT montrés ici. Pour mener à bien ce montage, vous aurez besoin d'un Raspberry Pi. Il sera question plus bas de commander une LED située sur le Raspberry Pi.

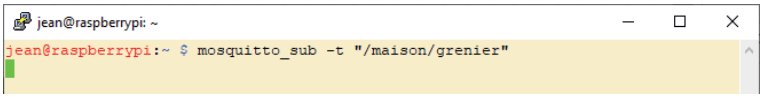
Examinons maintenant des exemples concrets sur le Raspberry Pi. Pour installer le logiciel, entrez les instructions suivantes dans une fenêtre de terminal :

```
# sudo apt-get update
# sudo apt-get install mosquitto mosquitto-clients
```

Pour permettre la réalisation de la connexion de l'ESP32 au serveur MQTT du Raspberry Pi, il est nécessaire de créer le fichier */etc/mosquitto/conf.d/mosquitto.conf* et d'y ajouter les paramètres de configuration suivants puis de redémarrer mosquitto :

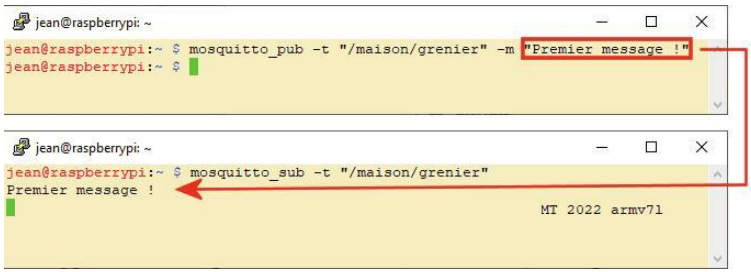


Aussitôt après l'installation, vous pouvez démarrer un premier test sur votre Raspberry Pi et vous assurer que MQTT fonctionne correctement. MQTT est immédiatement opérationnel. Nous devons d'abord expliquer dans MQTT quel Topic suivre à l'aide d'un abonnement. L'instruction `mosquitto_sub` permet de définir le Topic souhaité à l'aide de la clé `-t` :

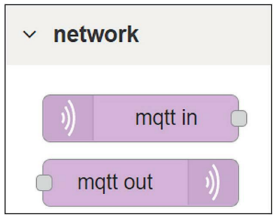


L'instruction donnée est simplement acquittée en plaçant le curseur sans invite à la ligne suivante. Vous devez ensuite ouvrir une deuxième fenêtre

de terminal (Ctrl+Alt+F2) pour publier un message à l'aide de l'instruction `mosquitto_pub` et de la clé `-m` :



Une fois la commande confirmée avec la touche Entrée, le message qui vient d'être saisi s'affiche dans la première fenêtre de terminal (Ctrl+Alt+F1). Vous venez de tester avec succès le Broker MQTT. Vous voudrez sans doute savoir ensuite si Node-RED est aussi en mesure d'échanger avec le Broker MQTT et si celui-ci peut recevoir en message envoyé. Node-RED fournit d'office un Node *mqtt-out* que vous trouverez dans la palette *network* :

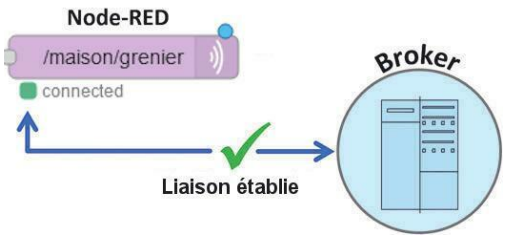


Il est alors très simple d'envoyer un message au Broker MQTT à l'aide d'un Node *Inject*. C'est ce que nous allons justement faire avec le Flow suivant :

Figure 27-7 ▶
Flow MQTT

La coche verte nous indique que la liaison du Node MQTT au Broker MQTT est établie :

Figure 27-8 ▶
Node MQTT connecté au Broker MQTT



Cliquez sur le Node *inject* pour envoyer le message « Bonjour, ici Node-RED ». La configuration du Node MQTT est aisée et se présente sous la forme suivante :

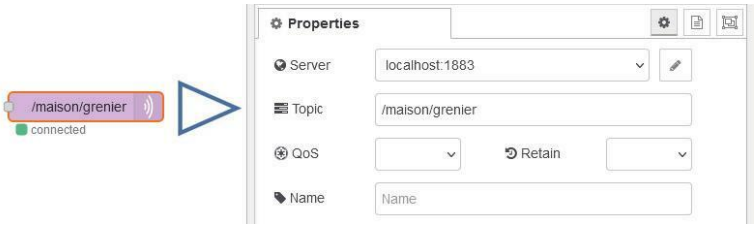


Figure 27-9
Configuration du Node mqtt

Comme le Broker MQTT et Node-RED fonctionnent sur le Raspberry, vous pouvez utiliser ici *localhost* au lieu de l'adresse IP du Raspberry Pi. N'oubliez pas d'ajouter le port 1883. Toutefois, si le Broker doit être adressé de l'extérieur, vous devrez indiquer obligatoirement une adresse IP unique. Nous reviendrons sur ce point. Le Topic inséré ici doit évidemment correspondre au Topic que nous avons utilisé pour l'abonnement. C'est effectivement le cas. Le message est envoyé après le déploiement du Flow et un clic sur le Node *inject*. Voyons voir quelle est la réaction dans notre fenêtre de terminal :

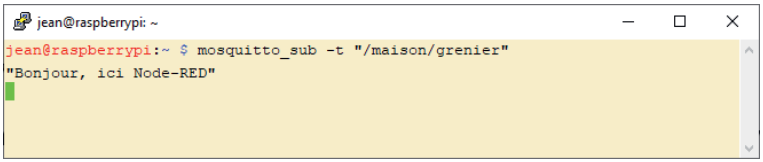
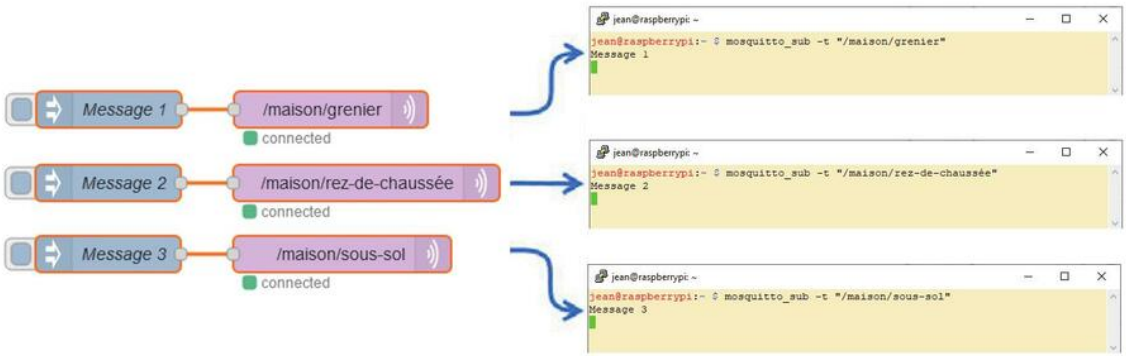


Figure 27-10
Message de Node-RED confirmé par le Broker MQTT

C'est précisément le message que nous souhaitions envoyer. Vous pouvez ainsi publier différents messages avec différents Topics ; ceux-ci apparaissent dans les fenêtres de terminal en fonction de l'abonnement correspondant. Faites donc le test suivant avec trois messages/Topics différents :

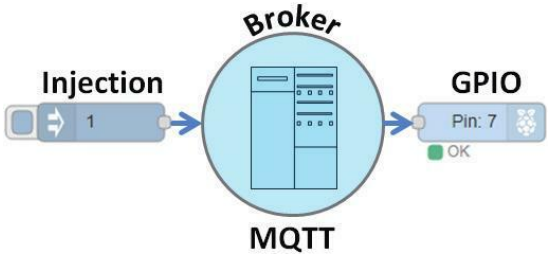
Figure 27-11
Différents messages envoyés de Node-RED



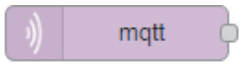
Un test rudimentaire

Mon objectif est de vous montrer avec ce petit test comment commander facilement une LED raccordée à une broche du GPIO sur le Raspberry Pi. Cela ne se fait pas directement, mais par l'intermédiaire de MQTT. Le signal de commande de la LED ne va pas directement à la LED mais est envoyé en direction de MQTT. Une fois sa réception validée, le message est ensuite dirigé de MQTT à la broche concernée. Le Broker MQTT sert donc d'instance de transfert entre le Node *inject* et la commande de la LED. Examinez le schéma suivant :

Figure 27-12 ▶
Le Broker MQTT commande la LED sur le Raspberry Pi.



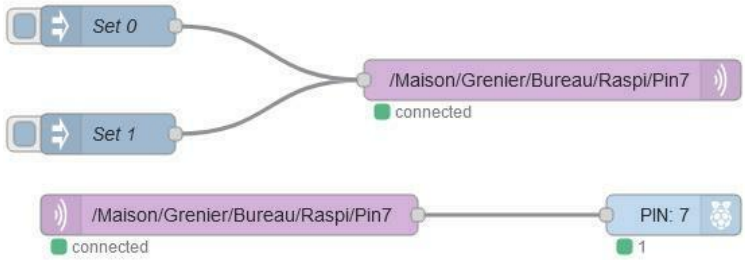
La mise en pratique de ce genre de Flow est très simple et se présente avec le schéma suivant. En plus du Node MQTT *Output* déjà vu, nous utilisons maintenant le Node MQTT *Input*, que vous trouverez également dans la palette *network* :



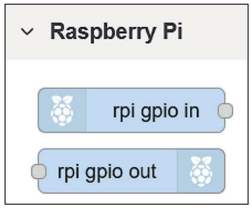
Ce Node sert à transférer les messages reçus.

Pour commander les broches GPIO du Raspberry, nous devons disposer de Nodes GPIO pour le Raspberry Pi qui doivent être installés à partir de la palette de Node-RED (menu *Manage palette*, onglet *Install*) en recherchant node-red-node-pi-gpio.

Figure 27-13 ▶
Commande d'une broche GPIO sur le Raspberry Pi via MQTT



Le Node GPIO pour le Raspberry Pi est fourni d'office dans le Node éponyme. Pour ce test, utilisons le Node *gpio out* :



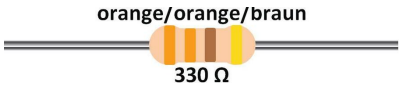
Pour commander le Node *gpio out*, nous ne pouvons pas recourir aux valeurs logiques *false* ou *true* mais aux valeurs numériques 0 ou 1. Pour ce test, j'utilise la broche 7 qui se cache sous la désignation GPIO04 dans le Raspberry Pi. La configuration du Node est facile à entreprendre, il existe en effet une vue d'ensemble de la disposition des différentes broches dans l'interface GPIO :

3.3V Power - 1	2 - 5V Power
SDA1 - GPIO02 - 3	4 - 5V Power
SCL1 - GPIO03 - 5	6 - Ground
GPIO04 - 7	8 - GPIO14 - TxD
Ground - 9	10 - GPIO15 - RxD
GPIO17 - 11	12 - GPIO18
GPIO27 - 13	14 - Ground
GPIO22 - 15	16 - GPIO23
3.3V Power - 17	18 - GPIO24
MOSI - GPIO10 - 19	20 - Ground
MISO - GPIO09 - 21	22 - GPIO25
SCLK - GPIO11 - 23	24 - GPIO8 - CE0
Ground - 25	26 - GPIO7 - CE1
SD - 27	28 - SC
GPIO05 - 29	30 - Ground
GPIO06 - 31	32 - GPIO12
GPIO13 - 33	34 - Ground
GPIO19 - 35	36 - GPIO16

Figure 27-14
Broches GPIO sur le Raspberry Pi

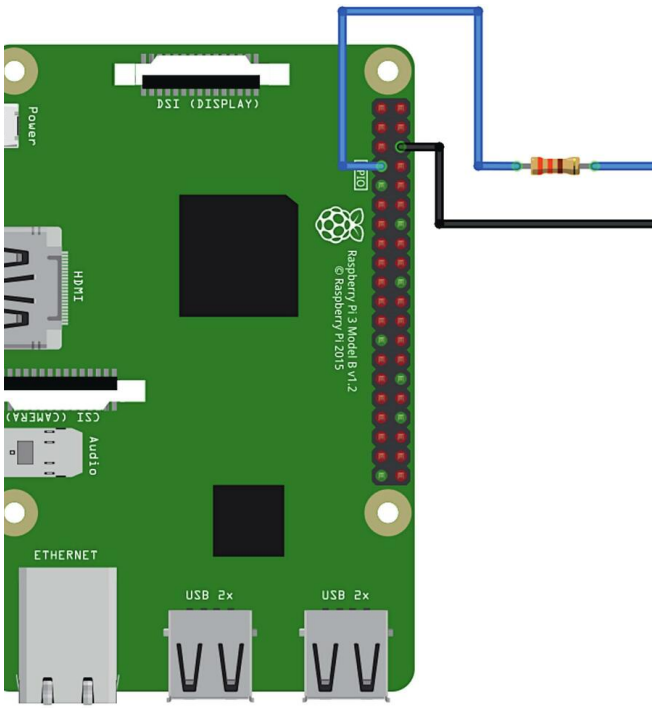
La connexion de la LED sur l'interface GPIO est facile. Vous devez pour autant veiller avec la plus grande attention à éviter les courts-circuits qui pourraient détruire le Raspberry Pi : les broches GPIO sont en effet reliées directement au processeur et ne tolèrent aucune négligence ! En plus du Raspberry Pi, vous aurez besoin des composants suivants :

Tableau 27-1 ▶
Liste des composants

Composants	
1 LED rouge	
1 résistance de 330 Ω	

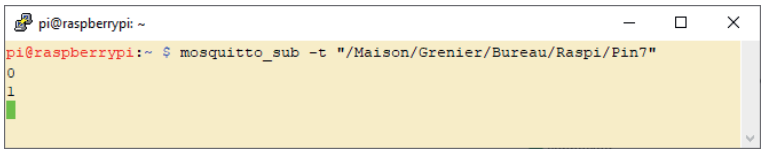
Le câblage se présente comme suit :

Figure 27-15 ▶
Câblage pour commander la LED sur le Raspberry Pi



Selon l'injection, vous voyez la réaction de la LED connectée à la broche 7. Mais on peut aussi afficher les messages dans une fenêtre de terminal :

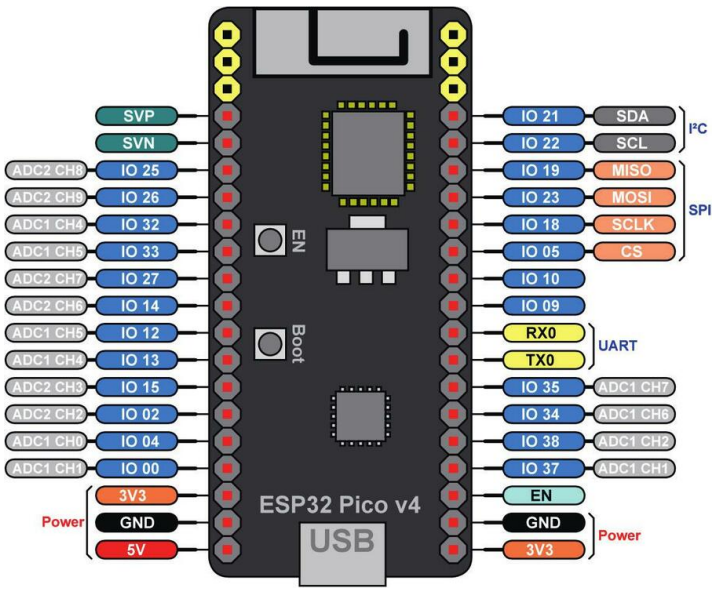
Figure 27-16 ▶
Messages de Node-RED pour commander la LED



Essayez maintenant de réaliser avec la configuration d'un Node *inject* un clignotement via MQTT.

Le module ESP32

Vous connaissez le module ESP32 du montage n° 22. Ce module est parfait pour établir des liaisons radio via Bluetooth ou Wi-Fi. Nous allons ici opter pour le Wi-Fi. Vous pouvez prendre n'importe quel ESP32 (pas forcément celui-ci). L'important est la broche Out. J'utilise pour ma part l'ESP32-Pico-Board avec la disposition de broches suivante :



◀ Figure 27-17
L'ESP32-Pico-Board

J'utilise aussi l'ESP32 Discoveryboard que j'ai construit et que je vous ai déjà présenté. L'intégration du support ESP32 dans l'environnement de développement Arduino a déjà été menée dans le montage ESP32 (montage n° 22). Si vous avez besoin de rafraîchir vos connaissances, n'hésitez pas à retourner jeter un coup d'œil.

ESP32 communique avec MQTT

Pour que votre ESP32-Pico-Board puisse communiquer avec MQTT, vous aurez besoin d'un client correspondant. Il se nomme *PubSubClient*. La bibliothèque associée peut être téléchargée à l'adresse Internet suivante :

https://github.com/knolleary/pubsubclient/archive/master.zip

Ajoutez ce fichier comprimé, sans extraire les fichiers, à l'environnement de développement Arduino à l'aide de l'option de menu *Sketch | Copier bibliothèque | Ajouter ZIP-Bibliothèque*. Une fois cette bibliothèque ins-

tallée, vous devriez voir les exemples ci-après en sélectionnant le menu *Fichier* | *Exemples* | *PubSubClient* :

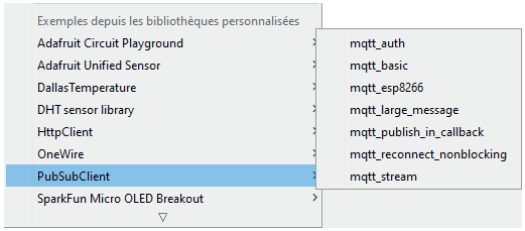


Figure 27-18 ▶
Exemples de bibliothèque PubSubClient

Je souhaite développer avec vous mon propre sketch, bien que les exemples de la bibliothèque soient aussi instructifs. Le test de fonctionnement de la connexion Wi-Fi vers votre routeur est le sketch que nous avons déjà vu dans le montage ESP32 (montage n° 22). Nous allons compléter le sketch avec la fonctionnalité MQTT en ajoutant l'adresse IP du Raspberry Pi et en intégrant la bibliothèque MQTT que nous venons de télécharger. Vous obtenez :

```
#include <WiFi.h> // WiFi client
#include <PubSubClient.h> // MQTT-PubSub-Client

const char* ssid = "EriksWifi";
const char* password = "SehrGeheim";
const char* mqtt_server = "192.168.0.26"; // IP du Raspberry Pi
long lastMsg = 0;
char msg[50]; // Message à envoyer
int value = 0; // Compteur de messages

WiFiClient espClient;           // WiFi-Client
PubSubClient client(espClient); // MQTT-Client

void setup() {
  Serial.begin(115200);

  WiFi.begin(ssid, password);
  Serial.println("Connection WiFi...");
  while(WiFi.status() != WL_CONNECTED) {
    delay(500);         // Courte pause
    Serial.print(".");  // Afficher des points en attente de connexion
  }
  Serial.println("\nConnecté au réseau WiFi");
  Serial.print("Adresse IP: ");
  Serial.println(WiFi.localIP());
  client.setServer(mqtt_server, 1883); // Initialisation du serveur MQTT
  client.setCallback(callback);        // Fonction pour messages entrant
  client.subscribe("/inTopic");        // Topic pour les messages d'entrée
}
```

La méthode `setCallback` fait référence à une fonction qui s'active à l'arrivée de messages. Vous pouvez définir avec la méthode `subscribe` quel Topic vous suivez. Dans notre cas, il s'agit du Topic *inTopic*.

```
void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  long now = millis();
  if(now - lastMsg > 2000) {
    lastMsg = now; ++value;
    snprintf(msg, 75, "Message #%ld envoyé par la carte ESP32", value);
    Serial.print("Publication du message: ");
    Serial.println(msg);             // Édition vers le moniteur série
    client.publish("outTopic", msg); // Publication du message dans MQTT
  }
}
```

Voici la définition de la fonction `callback` qui est activée à l'arrivée de messages relatifs au Topic indiqué :

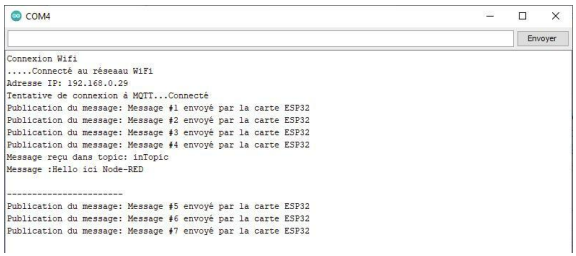
```
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message reçu dans topic: ");
  Serial.println(topic);
  Serial.print("Message:");
  for(int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  Serial.println("-----------------------");
}
```

La fonction `reconnect` s'active toujours en cas de problèmes de connexion. Elle permet de renouveler les initialisations de base :

```
void reconnect() {
  while(!client.connected()) {
    Serial.println("Tentative de connexion à MQTT...");
    // Creation d'un identificateur client aléatoire
    String clientId = "ESP32-Client-";
    clientId += String(random(0xffff), HEX);
    // Tentative de connexion
    if(client.connect(clientId.c_str())) {
      Serial.println("connecté");
      // Pubiler un message après connexion...
      client.publish("outTopic", "hello world");
      // ... et resouscription
      client.subscribe("inTopic");
    } else {
      Serial.print("échec, rc=");
      Serial.print(client.state());
      Serial.println(" nouvel essai dans 5 secondes");
      delay(500); // Pause de 5 secondes
      Serial.println("");
      delay(4500);
    }
  }
}
```

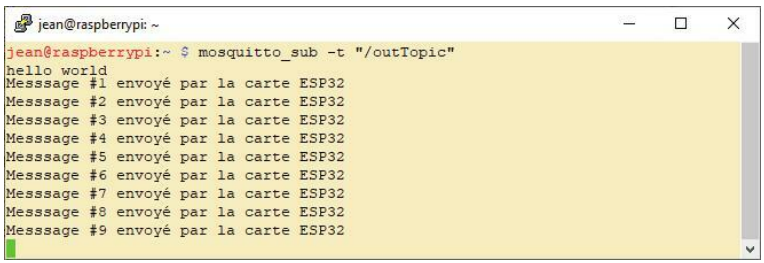
Le sketch utilise le Topic appelé *outTopic*, à qui un message est envoyé toutes les deux secondes. Ouvrez le moniteur série après le téléchargement du firmware pour voir si l'ESP32-Pico-Board s'est bien identifié sur votre routeur. Assurez-vous que la vitesse de transmission est réglée sur 115 200 bauds.

Figure 27-19 ▶ Message de l'ESP32-Pico-Board dans le moniteur série



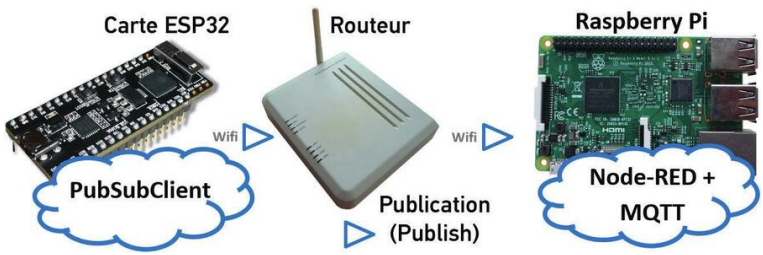
Cela se présente bien et j'ai pu dans la foulée établir une connexion au routeur. L'adresse IP affichée n'est pas celle du serveur MQTT mais l'adresse IP attribuée à l'ESP32-Pico-Board par le routeur via le protocole DHCP. Côté Raspberry Pi, nous nous abonnons évidemment au Topic appelé *outTopic* avec l'instruction suivante :

Figure 27-20 ▶ Le message de l'ESP32-Pico-Board arrive sur le Raspberry Pi.



Et vous voyez que les messages de l'ESP32-Pico-Board arrivent ici aussi. J'ai représenté sur l'image suivante les différentes instances de la transmission :

Figure 27-21 ▶ Workflow de messages du Node MCU-Board vers le Raspberry Pi



Vous pouvez aussi raccorder votre Raspberry Pi au routeur par un câble LAN. Le Wi-Fi vous donnera un peu plus de flexibilité, même si les distances sont plus limitées. Il en va de même dans la direction opposée, car

un PubSubClient est installé sur l'ESP32-Pico-Board et, comme son nom l'indique, celui-ci peut donc non seulement publier mais aussi souscrire un abonnement. Le firmware que nous avons téléchargé permet de déterminer l'*outTopic* ainsi qu'un *inTopic*. Vous êtes alors en mesure de recevoir des messages sur le Node MCU-Board provenant du Raspberry Pi. Pour cela, ouvrez le moniteur série de l'environnement de développement Arduino et donnez l'instruction suivante dans une fenêtre de terminal de votre Raspberry Pi :

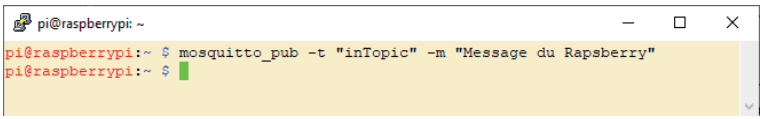


Figure 27-22
Message sur le Node MCU-Board provenant du Raspberry Pi

Le moniteur série indique parallèlement aux messages publiés le message du Raspberry Pi reçu :

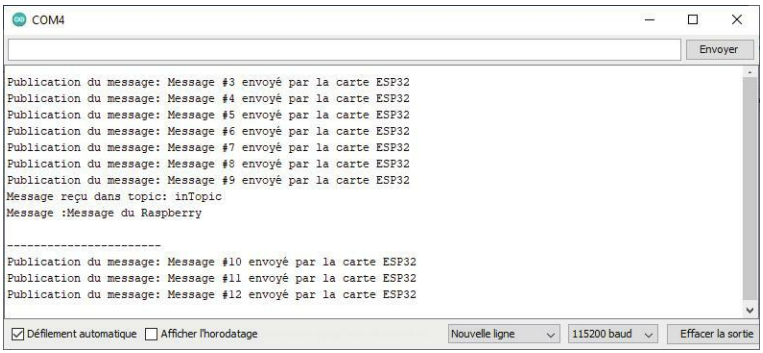


Figure 27-23
Message sur l'ESP32-Pico-Board dans le moniteur série de la carte Arduino-IDE

Vous voyez que le message a été reçu. Mettons maintenant tout cela en pratique dans Node-RED. Jusqu'à présent, nous n'avons utilisé qu'une fenêtre de terminal et le moniteur série comme instances d'entrée et d'affichage. Le Flow suivant nous permet d'afficher les messages d'*outTopic* dans la fenêtre Debug :

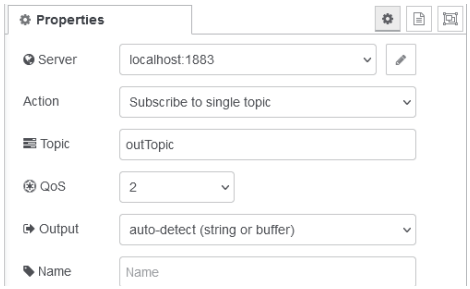
Figure 27-24
Flow MQTT pour la réception de messages

La fenêtre *debug* se présente comme suit :

Figure 27-265 ▶
Flow MQTT pour la réception de messages

Vous devez saisir dans le Node *MQTT-In* les paramètres suivants pour le serveur et le Topic :

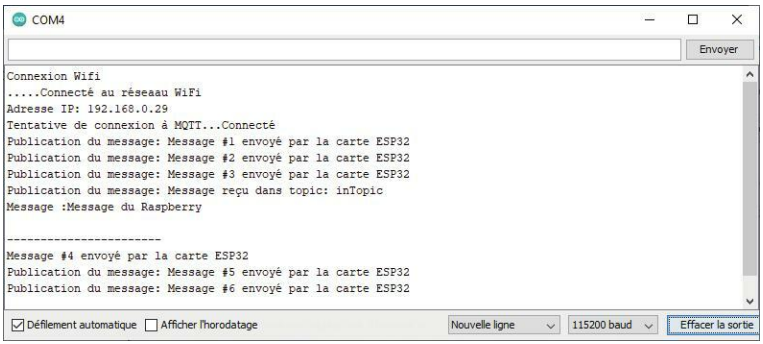
Figure 27-26 ▶
Configuration du Node MQTT-In



Il faudra faire la même chose dans la direction opposée. À vous de jouer ! Essayez de réaliser le Flow ci-après pour qu'un message provenant de Node-RED s'affiche sur le moniteur série Arduino. Le Flow est représenté ci-après. À votre tour de le configurer !

Figure 27-27 ▶
Envoi d'un message à partir de Node-RED

Vous voyez ci-après une représentation possible dans le moniteur série. Le message en provenance de Node-RED apparaît au milieu des messages à envoyer :



◀ Figure 27-28
Affichage du message provenant de Node-RED sur le moniteur série

Voilà pour les bases de MQTT et Node-RED.

Problèmes courants

Ce montage ne devrait pas poser de problèmes sur le plan matériel. Cependant, si quelque chose ne devait pas fonctionner, passez en revue les étapes de l'installation et de la configuration décrites dans ce montage. Cela devrait résoudre le problème.

Qu'avez-vous appris ?

Dans ce montage, vous avez découvert les bases MQTT et appris comment utiliser votre Raspberry Pi comme serveur MQTT. Vous avez suivi l'organisation des messages en Topics, vous savez maintenant comment publier des messages et s'y abonner. L'instance centrale pour gérer les messages est le Broker MQTT, situé entre les capteurs et les appareils terminaux. Node-RED fournit des Nodes « prêts à l'emploi » pour communiquer avec MQTT et nous avons envoyé des messages de Node-RED à une fenêtre de terminal dans le Raspberry Pi. Dans le sens opposé, des messages ont été envoyés du Raspberry Pi à Node-RED via une fenêtre de terminal. Vous avez également fait connaissance du module Wi-Fi ESP32 et appris à programmer avec l'environnement de développement Arduino à l'aide de l'extension correspondante.

Index

D

E

F

G

H

I

J

K

S

T

U

V

W