Christian Tavernier

Arduino

Maîtrisez sa programmation et ses cartes d'interface *(shields)*

2e édition

DUNOD

Tout le catalogue sur
www.dunod.com

ÉDITEUR DE SAVOIRS

Maquette de couverture : WIP

Illustrations intérieures : Alain et Ursula Bouteveille-Sanders

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2011, 2014
ISBN 978-2-10-071072-0

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

Avant-propos XI

1 • Qu'est-ce que l'Arduino ? 1

1.1 Qu'est-ce qu'un microcontrôleur ? 1
1.1.1 L'alimentation 2
1.1.2 L'horloge 2
1.1.3 Le circuit de reset 3
1.1.4 La programmation 3
1.2 Les différentes versions d'Arduino 4
1.3 L'Arduino Uno 5
1.3.1 Dimensions et organisation 5
1.3.2 Alimentation 6
1.3.3 Horloge 6
1.3.4 Reset 6
1.3.5 Les mémoires 7
1.3.6 Les entrées/sorties 8
1.3.7 Différences entre Arduino Uno Révision 1, 2 et 3 10
1.3.8 Schémas des Arduino Uno 10
1.4 L'Arduino Mega 2560 11
1.4.1 Les mémoires 12
1.4.2 Les entrées/sorties 12
1.4.3 Schéma de l'Arduino 2560 13
1.5 Langage machine et langage évolué 14
1.5.1 À l'origine : le langage machine 14
1.5.2 Les langages évolués 15
1.5.3 L'environnement de développement de l'Arduino 16
1.5.4 Le langage évolué de l'Arduino 18
1.6 L'environnement de développement 18
1.6.1 Installation de l'environnement de développement 19
1.6.2 Première configuration de l'environnement de développement 20

© Dunod – La photocopie non autorisée est un délit.

1.6.3 Utilisation de l'environnement de développement 21
1.6.4 Un premier programme juste pour « le fun » 23

2 • Environnement de développement et langage de programmation 25

2.1 Organisation générale d'un programme 26
2.1.1 Instructions et commentaires 26
2.1.2 Inclusions et définitions 27
2.1.3 Structure d'un programme 28
2.2 Données, variables et constantes 29
2.2.1 Les données numériques 30
2.2.2 Les données logiques et diverses 31
2.2.3 Les tableaux ou données indexées 32
2.2.4 Les données caractères et chaînes de caractères 32
2.2.5 Les conversions de types 35
2.2.6 Visibilité des variables 35
2.2.7 Gestion de la taille des variables 37
2.2.8 Constantes particulières 37
2.3 Fonctions arithmétiques et mathématiques 39
2.3.1 Opérateurs arithmétiques 39
2.3.2 Opérateurs arithmétiques composés 40
2.3.3 Fonctions mathématiques 41
2.3.4 Fonctions trigonométriques 42
2.4 Opérateurs de comparaison et opérateurs logiques 42
2.4.1 Opérateurs de comparaison 43
2.4.2 Opérateurs logiques booléens 43
2.4.3 Opérateurs logiques au niveau bit 44
2.5 Les structures de contrôle 46
2.5.1 Les prises de décision 46
2.5.2 Les boucles 50
2.5.3 Les sauts 51
2.6 Gestion du temps et des entrées/sorties 53
2.6.1 Gestion du temps 53
2.6.2 Entrées/sorties numériques 55
2.6.3 Entrées analogiques 57
2.6.4 Sorties analogiques 58
2.7 Entrées/sorties particulières 60
2.8 Fonctions diverses 62
2.8.1 Génération de nombres aléatoires 62

2.8.2 Manipulation de bits 63
2.8.3 Gestion des interruptions 65
2.9 Gestion du port série 67

3 • Les cartes d'interface ou shields de base 71

3.1 Les shields prototypes 71
3.1.1 Le minimum vital 72
3.1.2 Les éléments optionnels 73
3.1.3 Bonnes et mauvaises cartes prototypes 73
3.1.4 Un shield qui n'en est pas un 76
3.2 Un shield idéal pour la formation 77
3.3 Un shield vide mais fort utile 79
3.4 Un shield pour y voir clair 81

4 • Entrée/sorties numériques parallèles : poussoirs, claviers, LED, relais et afficheurs 85

4.1 Les entrées numériques parallèles 86
4.1.1 Poussoirs, commutateurs, capteurs de type tout ou rien 86
4.1.2 Comment vaincre les rebondissements ? 88
4.1.3 Claviers en matrices 90
4.1.4 Plusieurs touches avec une seule entrée 94
4.1.5 Entrées haute tension et opto-isolées 97
4.2 Les sorties numériques parallèles 98
4.2.1 Commandes de LED et de relais 99
4.2.2 Commande directe de charges alimentées en continu 102
4.2.3 Commande directe de charges reliées au secteur 103
4.2.4 Commande directe d'afficheurs à LED 104
4.2.5 Commande directe d'afficheurs à LED multiplexés 106
4.2.6 Commande d'afficheurs à LED avec des circuits spécialisés 109
4.2.7 Utilisation d'afficheurs à cristaux liquides (LCD) 109

5 • Entrées/sorties analogiques : capteurs, potentiomètres, gradateurs et moteurs 117

5.1 Les entrées analogiques 117
5.1.1 Utilisation de capteurs analogiques résistifs 118
5.1.2 Où il est question d'hystérésis 120
5.1.3 Utilisation de capteurs délivrant une tension analogique 121
5.1.4 Utilisation de capteurs délivrant « n'importe quoi » 124

© Dunod – La photocopie non autorisée est un délit.

5.2 Les sorties analogiques PWM **126**
5.2.1 Gradateur de tension ou de puissance PWM 126
5.2.2 Tension continue à partir d'un signal PWM 129
5.3 La commande des moteurs **130**
5.3.1 Les servos de radiocommande 130
5.3.2 Les moteurs pas à pas 137
5.3.3 Les moteurs à courant continu 142

6 • Les interfaces séries synchrones et asynchrones, I2C, SPI, bus « un fil » **147**
6.1 Les liaisons séries asynchrones **147**
6.1.1 Interface matérielle ou logicielle 147
6.1.2 L'interface RS 232 150
6.1.3 Des signaux de contrôle parfois inutiles 153
6.2 Les liaisons séries synchrones **154**
6.2.1 Le registre à décalage 154
6.2.2 Commande de LED ou d'afficheurs avec un registre à décalage 156
6.3 Le bus I2C **158**
6.3.1 Généralités 159
6.3.2 Protocole du bus I2C 159
6.3.3 La bibliothèque Wire de l'Arduino 162
6.3.4 Exemple d'utilisation du bus I2C 165
6.4 L'interface ou bus SPI **170**
6.4.1 Principe d'une liaison de type SPI 170
6.4.2 La bibliothèque SPI de l'Arduino 172
6.4.3 Exemple d'utilisation d'une liaison SPI 173
6.5 Le bus « un fil » de Dallas (Maxim) **177**
6.5.1 Principe du bus un fil 177
6.5.2 La bibliothèque OneWire de Jim Studt 179
6.5.3 Exemple d'utilisation du bus un fil 182

7 • Pour aller plus loin **185**
7.1 Les shields évolués **185**
7.1.1 Programmation des shields évolués 186
7.1.2 Reliez votre Arduino à Internet 186
7.2 Les autres Arduino originaux **192**
7.2.1 L'Arduino Mini 192
7.2.2 L'Arduino Nano 194
7.2.3 L'Arduino Fio 195
7.2.4 L'Arduino Pro 196

7.2.5 L'Arduino Pro Mini 198
7.2.6 L'Arduino LilyPad 199
7.3 Les « Arduino like » 200
7.3.1 Le Eleven, anciennement TwentyTen de Freetronics 201
7.3.2 Le Freeduino 202
7.3.3 Le Boarduino 203
7.4 Les « gros » Arduino 204
7.4.1 L'Arduino Leonardo 204
7.4.2 Les Arduino Due, Yun et Galileo 206

Webographie 207

Index 209

© Dunod – La photocopie non autorisée est un délit.

AVANT-PROPOS

L'Arduino, ou plutôt les Arduino devrions-nous écrire car ils ont fait des petits, connaissent aujourd'hui un succès planétaire, pourtant, il ne s'agit au départ que d'une simple carte électronique de petit format supportant un microcontrôleur, entouré du minimum de composants nécessaires à son fonctionnement de base. Rien de très original jusque-là nous direz-vous, et surtout rien qui puisse expliquer ce succès.

En fait, cette carte supporte également, sur sa périphérie, une rangée de connecteurs dans lesquels peuvent venir s'enficher directement des cartes d'interface, appelées les « shields ». Ces cartes d'interface sont aujourd'hui très nombreuses et couvrent quasiment tous les besoins des applications courantes.

Mais ce n'est pas tout. L'Arduino est également équipé d'un connecteur USB permettant de le raccorder à un micro-ordinateur, PC ou Mac peu importe, utilisé pour écrire le programme destiné à le piloter, mais également pour programmer le microcontrôleur qui équipe l'Arduino sans aucune intervention matérielle de votre part.

Cerise sur le gâteau, le projet Arduino dans son ensemble, c'est-à-dire les cartes mais aussi les logiciels associés, sont « open source » c'est-à-dire sont publics, libres de droits et donc utilisables par tous sans devoir verser de quelconques royalties ou droits d'auteur.

En d'autres termes, l'Arduino permet à tout un chacun de réaliser à moindre coût des applications à base de microcontrôleur quasiment sans nécessiter de connaissances en électronique, sans avoir besoin de sortir le fer à souder et sans devoir investir dans des outillages spécialisés tels qu'outils de développement ou programmateurs par exemple.

C'est en grande partie à cette conjonction de facteurs que l'Arduino doit son succès, et nous allons voir, dans la suite de cet ouvrage, qu'il n'est pas usurpé.

■ Avertissement relatif aux schémas

À partir du chapitre 4, et jusqu'à la fin de cet ouvrage, vous allez pouvoir découvrir un certain nombre de schémas d'applications ou d'interfaces mettant en œuvre l'Arduino. Compte tenu de la relative universalité des entrées/sorties de ce dernier, nous avons adopté, les conventions suivantes.

Si, sur un schéma, ne figure aucune indication de nom ou de numéro de ligne d'entrée/sortie, cela signifie que n'importe quelle entrée/sortie numérique de

© Dunod – La photocopie non autorisée est un délit.

l'Arduino peut être utilisée, sans restriction. Sur l'Arduino Uno, cela signifie donc que l'on peut utiliser les entrées/sorties 2 à 13 incluses et sur l'Arduino Mega 2560 que l'on peut utiliser les entrées/sorties 2 à 53 incluses.

Si un numéro de ligne est indiqué, cela signifie :

– soit que le choix de la ligne est imposé parce qu'elle supporte une interface spécifique (Rx, Tx, SDA, SCL, /SS, MISO, MOSI, SCK, PWM) mais cela est alors clairement indiqué dans le texte associé ;
– soit que c'est cette ligne qui a été utilisée dans le programme associé au schéma.

Lorsque l'on doit utiliser une entrée analogique, celle-ci est toujours explicitement désignée sur les schémas afin que vous ne puissiez pas la confondre avec une entrée/sortie numérique, sous la forme AX où X est le numéro de l'entrée choisie. X peut varier de 0 à 5 pour l'Arduino Uno et de 0 à 15 pour l'Arduino Mega 2560.

Les entrées analogiques sont toutes échangeables entre elles sans restriction. Ainsi, un schéma utilisant par exemple A0, fonctionnera tout aussi bien si vous utilisez A1. Il faudra juste éventuellement penser à modifier le programme associé en conséquence.

■ Compléments en ligne

Vous trouverez sur *www.dunod.com/contenus-complementaire/9782100710409* et sur *www.tavernier-c.com*, à la page consacrée à cet ouvrage, un certain nombre de compléments organisés comme indiqué ci-après.

□ Répertoire fiches techniques

Ce répertoire contient les fiches techniques au format pdf de tous les circuits intégrés et composants particuliers utilisés sur les schémas présentés dans cet ouvrage. Vous pouvez vous y reporter pour connaître précisément leurs caractéristiques électriques et les limites à ne pas dépasser, ou bien encore pour découvrir leurs registres internes et la signification de leurs contenus, notamment pour ce qui est des circuits disposant d'une interface SPI, I2C ou bien encore bus un fil. Chaque fichier porte le nom du composant ou de la famille de composants qu'il concerne.

□ Répertoire schémas cartes

Ce répertoire contient les schémas de toutes les versions de cartes Arduino ainsi que les schémas des différents shields présentés ou utilisés dans l'ouvrage. Ces schémas sont directement issus de la documentation de leurs fabricants respectifs et sont disponibles sous forme de fichiers au format pdf. Les noms donnés aux fichiers sont sans ambiguïté quant aux schémas qu'ils contiennent.

□ Répertoire bibliothèques

Par précaution, vous trouverez dans ce répertoire les bibliothèques utilisées dans certains programmes de l'ouvrage ; bibliothèques qui sont proposées sur des sites autres que le site officiel consacré à l'Arduino et dont la pérennité ne peut donc être assurée.

□ **Répertoire programmes**

Comme son nom l'indique, ce répertoire contient les listings de tous les programmes présentés dans l'ouvrage. Ils sont tous au format ino, prêts à être compilés, et peuvent donc être copiés directement dans la fenêtre de l'éditeur de l'outil de développement de l'Arduino.

Ces programmes sont directement issus du PC de l'auteur ayant servi à leur développement et sont donc parfaitement conformes aux versions qui sont présentées dans cet ouvrage.

Les noms de fichiers correspondent aux repères donnés aux listings dans l'ouvrage (L61.pde pour le listing 6.1 par exemple).

■ **Avertissement relatif aux versions d'Arduino**

L'Arduino est un produit qui évolue très vite, trop même diront certains, et, même si cet ouvrage est régulièrement mis à jour, de nouvelles versions d'Arduino auront peut-être été commercialisées entre la dernière mise à jour de ce livre et son achat par vos soins.

Afin que vous puissiez néanmoins utiliser avec profit nos exemples de schémas et de programmes, nous avons donc fait le choix d'utiliser pour vous les présenter l'Arduino Uno à partir de la Révision 3. Toutes les versions ultérieures d'Arduino Uno sont en effet « upward compatible » de cette version, comme disent nos amis anglo-saxons, c'est-à-dire que, même si les versions ultérieures comportent des fonctions ou des possibilités supplémentaires, elles restent totalement compatibles des versions précédentes.

Sachez aussi que, vu le succès rencontré par l'Arduino, d'autres versions que l'Arduino « de base » ou Arduino Uno sont aujourd'hui commercialisées. On trouve ainsi sur le marché l'Arduino Leonardo, l'Arduino Due et l'Arduino Tre ou Galileo est même en cours de commercialisation au moment où ces lignes sont écrites.

Ces versions d'Arduino sont des évolutions du modèle de base, visant à lui offrir plus de puissance et plus de ressources internes ou d'entrées/sorties mais tout ce que vous aurez appris dans cet ouvrage avec l'Arduino Uno sera évidemment utilisable avec ces nouvelles versions.

L'outil de développement de l'Arduino est lui aussi pris de la même frénésie d'évolution et son numéro de version ne cesse d'augmenter au fil du temps mais, là aussi, les programmes développés avec les versions précédentes peuvent être exploités sans aucun problème avec les versions actuelles.

1 • QU'EST-CE QUE L'ARDUINO ?

L'avant-propos de cet ouvrage a déjà un peu levé le voile : l'Arduino est une carte de circuit imprimé supportant un microcontrôleur et la circuiterie minimum nécessaire pour lui permettre de fonctionner, associée à une interface USB permettant de le programmer.

Nous allons détailler cela tout au long de ce chapitre et voir tout à la fois, ce qu'il y a réellement sur une carte Arduino, et le matériel nécessaire au développement de notre première application.

1.1 Qu'est-ce qu'un microcontrôleur ?

Pour faire simple, un microcontrôleur, appelé à tort microprocesseur par nombre de personnes mal informées, voire même par certains journalistes « scientifiques », est en fait l'équivalent d'un petit ordinateur, tel votre PC par exemple, contenu dans un seul boîtier de circuit intégré à plus ou moins grand nombre de pattes.

Il contient ainsi une unité centrale – l'équivalent du microprocesseur qui équipe votre PC – de la mémoire vive, de la mémoire morte, des interfaces diverses pour communiquer avec le monde extérieur et toute la circuiterie électronique et logique nécessaire pour faire fonctionner tout cela ensemble.

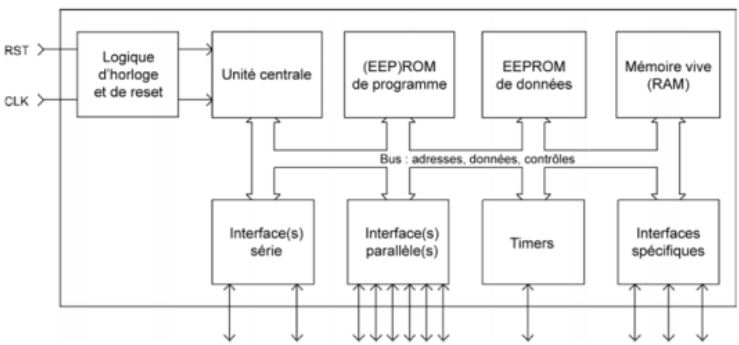

Figure 1.1 – Schéma simplifié du contenu type d'un microcontrôleur.

© Dunod – La photocopie non autorisée est un délit.

La figure 1.1 présente ainsi le contenu, très simplifié, mais largement suffisant pour l'instant, d'un microcontrôleur. On y constate que tous les éléments contenus dans le boîtier sont reliés entre eux par ce qui s'appelle un bus, qui est en fait un ensemble de connexions véhiculant les adresses, les données et les signaux de contrôle échangés entre ces différents sous-ensembles.

Pour ce qui est de ces sous-ensembles internes, leur nombre et leurs types sont variables et dépendent du microcontrôleur choisi. L'unité centrale est évidemment toujours présente puisque c'est elle le cerveau du microcontrôleur. La mémoire également puisqu'elle est indispensable pour contenir le programme que va exécuter le circuit, mais son type et sa taille varient énormément d'un circuit à un autre ; nous en reparlerons lorsque nous évoquerons les différentes versions d'Arduino.

Pour ce qui est des sous-ensembles d'interface, leur nombre et leurs types varient là aussi selon le microcontrôleur choisi, mais l'on rencontre quasiment toujours aujourd'hui un ou plusieurs *timers* ou compteurs, des entrées/sorties parallèles, des entrées/sorties séries et des convertisseurs analogiques/numériques et numériques/analogiques. Si ces termes vous sont totalement ou partiellement inconnus, ne vous inquiétez pas, nous y reviendrons dans la suite de cet ouvrage.

Compte tenu de l'intégration de tous ces éléments dans un seul et unique boîtier de circuit intégré, il ne faut que très peu de composants électroniques externes autour d'un microcontrôleur pour le faire fonctionner. Nous allons en dresser la liste rapidement.

1.1.1 L'alimentation

L'alimentation est évidemment le premier élément indispensable. Tous les microcontrôleurs actuels fonctionnent sous une tension unique qui varie, selon les types de circuits, de 1,8 à 6 volts environ, avec une prédilection encore aujourd'hui pour la tension de 5 volts.

Cette tension est en effet celle qui est utilisée par tous les circuits logiques de la famille dite TTL ; circuits qui ont été les premiers circuits logiques très largement utilisés et qui sont encore aujourd'hui très présents dans la majorité des appareils électroniques, même si on se dirige peu à peu vers des versions fonctionnant sous une tension de 3,3 volts.

De ce fait, lorsque l'on parle encore aujourd'hui de circuits logiques TTL ou compatibles TTL, cela sous-entend une alimentation sous 5 volts et, comme nous le verrons dans la suite de cet ouvrage, cela a également des implications sur les niveaux électriques admis en entrée et générés en sortie par ces circuits.

Quoi qu'il en soit, et quelle que soit sa valeur, la tension qui alimente notre microcontrôleur doit être stabilisée afin d'assurer un fonctionnement correct de celui-ci.

1.1.2 L'horloge

Le deuxième élément essentiel au fonctionnement d'un microcontrôleur est l'horloge. Un microcontrôleur est en effet un circuit logique séquentiel, c'est-à-dire qu'il fonctionne au rythme d'un signal rectangulaire, appelé horloge, qui cadence

toute sa circuiterie interne. La fréquence de cette horloge dépend du type de microcontrôleur utilisé et peut varier de quelques kilohertz (kHz) à plusieurs dizaines de megahertz (MHz) (dans un PC ou un Mac on dépasse même le gigahertz (GHz)). Plus la fréquence d'horloge est élevée, plus le microcontrôleur est rapide mais, en contrepartie, plus il consomme d'énergie.

Même s'il n'est pas indispensable que la fréquence de cette horloge soit parfaitement stable, c'est cependant généralement le cas car rares sont les applications qui n'ont pas besoin de mesurer du temps ou de générer des signaux aux chronogrammes précis, et cela n'est possible que si l'horloge qui pilote le microcontrôleur est elle-même stable. De ce fait, elle est en général pilotée par un quartz ou au moyen d'un résonateur céramique, seuls composants électroniques capables de générer des signaux à une fréquence stable et précise.

1.1.3 Le circuit de reset

Tout comme votre PC, un microcontrôleur exécute en permanence un programme et si, lorsque ce programme est au point et ne comporte plus de « bug » (oui, je sais, ce n'est jamais le cas sur votre PC…) il se déroule en continu tant que l'application est alimentée ; en phase de développement il est fréquent qu'il « se plante ». Pour sortir d'une telle situation, il est bien sûr possible de couper l'alimentation puis de la rétablir, ce qui fait redémarrer le programme au début. En effet, le microcontrôleur dispose d'un circuit de reset automatique à la mise sous tension qui se charge donc d'une telle opération.

Mais il est nettement plus confortable de pouvoir réaliser cette opération par simple appui sur un poussoir, et c'est là le rôle joué par le circuit de reset manuel. Lorsque l'on actionne le poussoir de reset, et quel que soit l'état dans lequel se trouve le microcontrôleur à ce moment-là, il recommence l'exécution du programme contenu dans sa mémoire à son début. S'il était « planté » il quitte donc cet état peu recommandable et reprend l'exécution du programme. S'il ne l'était pas, le poussoir de reset a bien évidemment le même effet.

1.1.4 La programmation

Un microcontrôleur, nous l'avons dit, exécute un programme Ce programme est contenu dans une de ses mémoires, qui peut être de la mémoire morte ou ROM, c'est-à-dire de la mémoire programmée une fois pour toutes par le fabricant du circuit, ou de la mémoire Flash, c'est-à-dire de la mémoire que l'utilisateur peut programmer et effacer tout à loisir.

Les microcontrôleurs à mémoire morte ne sont pas intéressants pour les développeurs d'applications car le programme qu'ils contiennent est figé une fois pour toutes lors de leur fabrication. On les rencontre donc seulement dans les appareils produits en très grande série au plan industriel. Le calculateur qui gère l'injection électronique d'un moteur de voiture est ainsi souvent un circuit de ce type car, une fois que son programme a été mis au point par le constructeur, il n'a plus besoin d'être modifié.

© Dunod – La photocopie non autorisée est un délit.

Les microcontrôleurs à mémoire Flash par contre représentent la solution idéale pour les développeurs d'applications. Cette mémoire, identique dans son principe à celle qui se trouve dans les cartes mémoires d'appareils photos ou bien encore dans les clés USB, peut en effet être programmée et effacée très simplement et, en outre, elle conserve son contenu lorsqu'elle n'est pas alimentée, ce qui est indispensable si l'on ne veut pas que notre microcontrôleur « oublie » son programme à chaque fois qu'on l'éteint !

Pour programmer cette mémoire, certains microcontrôleurs nécessitent d'être enlevés de leur application pour être placés sur un programmateur spécialement prévu à cet effet. Il s'agit là de la programmation classique ou traditionnelle.

D'autres circuits, plus récents et plus performants, supportent ce que l'on appelle la programmation « en circuit » ou ISP (*In System Programming*) ou bien encore ICSP (*In Circuit Serial Programming*) qui permet de les programmer, via une liaison spéciale, même lorsqu'ils restent en place sur leur application définitive. C'est évidemment beaucoup plus confortable que la programmation classique évoquée précédemment.

1.2 Les différentes versions d'Arduino

Maintenant que vous savez ce qu'est un microcontrôleur et ce qu'il faut pour pouvoir le faire fonctionner, nous pouvons passer à la présentation de l'objet de ce livre : l'Arduino ou, plus exactement, les différentes versions d'Arduino.

Les concepteurs de l'Arduino ont fixé leur choix sur les microcontrôleurs fabriqués par la société Atmel, et plus précisément sur la famille AVR de ce fabricant. Même si l'on peut discuter pendant des heures de ce choix et des mérites comparés de ces circuits par rapport à ceux de fabricants concurrents ; les microcontrôleurs AVR représentent aujourd'hui un excellent compromis prix/puissance et, même s'il ne s'agit que de microcontrôleurs à unité centrale 8 bits, ils permettent de développer des applications très intéressantes.

Le projet Arduino n'étant pas né en un jour, de nombreuses versions de cartes ont existé ou existent encore à ce jour mais, actuellement, on peut raisonnablement se focaliser sur les deux versions de base les plus récentes, tout en sachant qu'elles sont toutes *« upward compatibles »* comme disent les Américains, c'est-à-dire encore qu'elles présentent une compatibilité ascendante. Ainsi, une application développée pour une « vieille » carte Arduino Diecimila fonctionnera sans modification sur une récente Arduino Uno Révision 3 ou bien encore sur une Arduino Duemilanove. L'inverse, par contre, ne sera pas toujours vrai car les cartes récentes utilisent des microcontrôleurs disposant de mémoires de tailles plus importantes que leurs prédécesseurs.

À ce stade de l'ouvrage, on peut considérer aujourd'hui qu'il n'existe plus que deux versions majeures d'Arduino : l'Arduino Uno qui en est à sa Révision 3 au moment où ces lignes sont écrites, et l'Arduino Mega 2560 d'autre part, qui est une version compatible l'Arduino Uno mais disposant de ressources d'entrées/sorties beaucoup plus nombreuses.

Nous verrons, à la fin du chapitre 7, des versions moins conventionnelles de l'Arduino et de ses concurrents, et vous indiquerons alors ce que vous serez en droit d'en attendre.

1.3 L'Arduino Uno

Si vous débutez avec un Arduino, vous utiliserez nécessairement cette carte car elle est très peu coûteuse (environ 25 euros au moment où ces lignes sont écrites) et, à moins que votre application n'ait des besoins très particuliers nécessitant le recours à un Arduino Mega 2560, elle peut couvrir la majorité des besoins.

Bien que la version actuelle de cette carte en soit à sa Révision 3 au moment où ces lignes sont écrites, de nombreuses cartes en Révision 1 ou 2 sont encore en circulation et nous les traiterons donc toutes ensemble ci-dessous, étant entendu que la Révision 3 ne diffère des précédentes que par quelques points de détail que nous mettrons en évidence lorsque ce sera nécessaire.

1.3.1 Dimensions et organisation

Ces cartes sont au format « standard » Arduino c'est-à-dire encore qu'elles mesurent environ 52 mm sur 65 mm et se présentent comme indiqué figure 1.2.

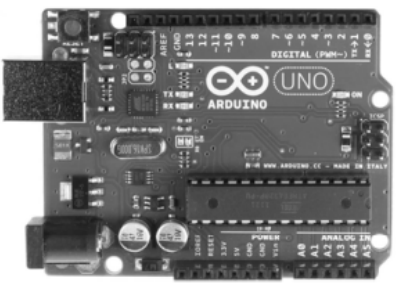

Figure 1.2 – La carte Arduino Uno Révision 3 (doc. *www.arduino.cc*).

Elles sont aujourd'hui toutes équipées d'un microcontrôleur Atmel ATmega 328 alors que, par le passé, les premières Duemilanove commercialisées n'embarquaient qu'un ATmega 168 dont la taille de la mémoire de programme est la moitié de celle du 328. Attention donc si vous « récupérez » une vieille Duemilanove à bien vérifier le type du processeur qui l'équipe (c'est écrit dessus !).

Deux rangées de connecteurs, situées de part et d'autre de la carte, permettent sa connexion au monde extérieur et, si l'on utilise les cartes d'interface, ou shields, au format Arduino, il est possible d'enficher directement ces derniers dans la carte de base, sur plusieurs niveaux si nécessaire. On réalise ainsi un module très compact et ne nécessitant quasiment aucune liaison par fils ; nous y reviendrons dans la suite de cet ouvrage.

© Dunod – La photocopie non autorisée est un délit.

1.3.2 Alimentation

L'alimentation du microcontrôleur qui équipe les cartes a lieu sous une tension de 5 volts qui peut provenir soit de la prise USB dont elles sont munies, ce qui est le cas lorsque la carte est reliée à un ordinateur ; soit d'un bloc secteur externe via le jack standard également présent sur la carte. La tension ainsi fournie doit alors être comprise entre 7 et 12 volts mais n'a pas besoin d'être stabilisée car un régulateur de tension prend place sur les cartes.

Contrairement aux premières versions d'Arduino, aucune commutation manuelle n'est nécessaire entre les deux sources d'alimentation ; celle-ci étant désormais automatique et transparente pour l'utilisateur.

Par ailleurs, et par mesure de sécurité pour l'ordinateur auquel sera relié l'Arduino, un fusible réarmable ou Polyfuse® est présent sur la connexion d'alimentation 5 volts de la prise USB. Toute consommation excessive sur cette dernière, c'est-à-dire en pratique supérieure à 500 mA, provoque le déclenchement de ce fusible, protégeant ainsi le port USB de l'ordinateur auquel la carte est reliée. Le fusible étant de type réarmable, il n'a pas besoin d'être remplacé lorsqu'il a été activé et il suffit de patienter quelques secondes pour qu'il retrouve son état normal.

Deux tensions stabilisées sont générées par les régulateurs montés sur les cartes Arduino : 5 volts et 3,3 volts et ces deux tensions sont disponibles sur les connecteurs placés sur le pourtour des cartes pour alimenter les shields (les cartes d'interface). Attention cependant à ne pas dépasser le courant maximum que peuvent fournir ces régulateurs. Il n'est ainsi que de 50 mA pour la sortie 3,3 volts et n'est pas précisé par le fabricant pour la sortie 5 volts.

1.3.3 Horloge

L'horloge est pilotée par quartz et fonctionne à la fréquence de 16 MHz ce qui confère à l'ATmega qui équipe les cartes des performances raisonnables. En pratique, vous n'aurez quasiment jamais à vous soucier de sa fréquence car les instructions du langage de programmation qui permet de programmer l'Arduino la prennent en compte automatiquement.

Ainsi par exemple, la fonction `delay(xxx)`, qui génère un délai de xxx ms, sait que l'horloge fonctionne à 16 MHz et fait donc automatiquement appel aux instructions nécessaires pour générer le délai que vous avez réellement demandé dans votre programme.

1.3.4 Reset

Toutes les cartes Arduino actuelles sont équipées d'un poussoir de reset manuel. Un appui sur celui-ci permet donc de relancer l'exécution d'un programme si nécessaire, soit parce qu'il s'est « planté » soit tout simplement parce que l'on souhaite le faire repartir de son début.

Mais il existe aussi sur les cartes Arduino deux autres sources de reset. La première est un reset automatique à la mise sous tension qui permet ainsi au programme

contenu en mémoire du microcontrôleur de démarrer automatiquement dès que la carte Arduino est alimentée. C'est bien évidemment une fonction indispensable dans toute application qui se respecte puisque celle-ci doit démarrer sans nécessiter l'intervention de l'utilisateur à chaque mise sous tension.

L'autre source de reset est plus « discrète » est n'est généralement utilisée qu'en phase de développement de programme. En effet, lorsque l'on a terminé l'écriture d'un programme destiné à l'Arduino sur le micro-ordinateur utilisé pour cela (PC ou Mac peu importe) il faut programmer la mémoire du microcontrôleur de l'Arduino avec ce dernier. Pour cela, on fait appel à un petit programme spécial, appelé un « *bootloader* », déjà contenu dans la mémoire du microcontrôleur. Cela fonctionne de la façon suivante.

Suite à un reset, le *bootloader* est activé et attend de recevoir des informations de programmation via le port USB de l'Arduino. S'il ne les reçoit pas, il lance l'exécution du programme déjà contenu en mémoire du microcontrôleur ce qui fait démarrer l'application correspondante. S'il les reçoit, il utilise alors les données qui lui parviennent ensuite via le port USB pour programmer la mémoire de programme du microcontrôleur.

Lorsque l'on est en phase de développement de programme, on effectue très souvent des opérations de programmation de la mémoire du microcontrôleur afin de tester le programme que l'on écrit au fur et à mesure de son avancement. Pour ne pas devoir à chaque fois appuyer sur le poussoir de reset puis lancer manuellement la programmation, le reset peut être provoqué directement, depuis le logiciel de développement et via le port USB, chaque fois que l'on souhaite programmer la mémoire du microcontrôleur. C'est donc là sa troisième source de reset.

1.3.5 Les mémoires

L'ATmega 328 qui équipe l'Arduino dispose de 32 kilo-octets de mémoire de programme. Cette mémoire est de type Flash, analogue à celle que l'on trouve par exemple dans les clés USB. Elle se programme électriquement, de manière transparente pour vous lorsque l'Arduino est utilisé avec son logiciel de développement fonctionnant sous PC ou Mac, au travers du connecteur USB qui équipe les Arduino et grâce au « *bootloader* » qui est contenu d'origine dans le microcontrôleur. Une fois programmée, cette mémoire Flash conserve son contenu indéfiniment, même lorsque la carte n'est pas alimentée.

L'ATmega 328 contient aussi de la mémoire vive ou RAM, analogue dans son principe à la mémoire vive de vos PC ou Mac, mais en beaucoup plus petite quantité puisque l'on ne dispose ici que de 2 kilo-octets (contre plusieurs Giga octets dans un PC par exemple). Cette mémoire est généralement utilisée pour les variables employées dans les programmes, pour stocker des résultats temporaires lors de calculs, etc. Elle présente la particularité de pouvoir être lue et écrite à tout instant par le microcontrôleur mais elle « oublie » son contenu dès qu'il n'est plus alimenté. Il est donc hors de question de l'utiliser pour stocker des informations qui devront être conservées lorsque la carte ne sera pas alimentée.

© Dunod – La photocopie non autorisée est un délit.

Pour cela, les ATmega 328 disposent également de mémoire EEPROM, acronyme qui signifie mémoire programmable et effaçable électriquement. La taille de cette EEPROM est seulement de 1 kilo-octets mais, vu son rôle, c'est généralement suffisant.

Cette mémoire EEPROM est une mémoire dans laquelle le microcontrôleur peut lire à tout instant. Il peut aussi y écrire et effacer son contenu, avec plus de facilité que dans la mémoire Flash de programme, mais avec moins de facilité que dans la mémoire vive. En contrepartie, et comme c'est le cas pour la mémoire Flash de programme, le contenu de l'EEPROM est conservé même lorsqu'elle n'est plus alimentée. C'est donc une mémoire qui sera utilisée pour conserver des données ou des paramètres que l'on doit pouvoir retrouver d'une utilisation à l'autre de l'Arduino.

Pour faire un parallèle avec des produits que vous connaissez, c'est par exemple dans une mémoire de ce type que votre téléviseur stocke les informations relatives aux chaînes de la TNT que vous lui faites découvrir lorsque vous le mettez en marche pour la première fois. Il conserve ensuite ces informations même lorsqu'il est éteint mais, si vous changez de région, vous pourrez à nouveau lui faire mémoriser de nouvelles informations qui remplaceront alors les précédentes dans son EEPROM.

1.3.6 Les entrées/sorties

Les entrées/sorties de l'Arduino sont les éléments avec lesquels vous serez le plus souvent en contact. En effet, c'est par leur intermédiaire que le microcontrôleur de l'Arduino communique avec le monde extérieur et donc avec les cartes d'interface ou shields, que ce soient des modèles commerciaux ou des cartes de votre cru. Voici donc ce qui est à votre disposition pour cela sur les Arduino Uno.

■ Entrées/sorties numériques

Quatorze lignes d'entrées/sorties numériques parallèles, repérées 0 à 13, sont disponibles ; chacune d'entre elles pouvant fonctionner en entrée ou en sortie sous le contrôle du programme et ce sens de fonctionnement pouvant même changer de manière dynamique pendant son exécution. Ces entrées/sorties admettent et délivrent des signaux logiques compatibles TTL c'est-à-dire compris entre 0 et 5 volts. Sans l'aide d'aucun composant extérieur (voir pour cela les chapitres 4 et suivants), elles peuvent absorber ou délivrer un courant maximum de 40 mA.

> **Attention !**
> Même si chaque sortie peut théoriquement fournir ou absorber jusqu'à 40 mA, le courant total fourni ou absorbé par l'ensemble des sorties ne saurait en aucun cas dépasser 200 mA.

Ces lignes partagent cependant leur rôle avec certaines interfaces spécialisées contenues dans le microcontrôleur. Si ces interfaces ne sont pas utilisées par l'application, les entrées/sorties parallèles fonctionnent comme telles. Si ces interfaces sont utilisées par l'application, les lignes d'entrées/sorties parallèles qui sont partagées avec elles ne sont évidemment plus disponibles en tant que telles.

Sous ces conditions de partage on peut ainsi disposer de :

- une entrée (*Rx*) et une sortie (*Tx*) série asynchrone, partagées respectivement avec les lignes 0 et 1 ;
- deux entrées d'interruptions externes partagées avec les lignes 2 et 3 ;
- six sorties PWM (*Pulse Width Modulation* ou modulation de largeur d'impulsions) partagées avec les lignes 3, 5, 6, 9, 10 et 11 ;
- quatre entrées/sorties d'interface avec le bus série normalisé de type SPI réparties comme suit ; /SS sur 10, MOSI sur 11, MISO sur 12 et SCK sur 13 ;
- deux entrées/sorties d'interface série I2C réparties comme suit : SDA sur A4 et SCL sur A5 ;
- depuis la Révision 3 de l'Arduino, les lignes SDA et SCL du bus I2C sont également disponibles sur leurs propres pinoches du connecteur destiné aux cartes d'interface mais attention, cela ne signifie pas que SDA et SCL sont devenues indépendantes de A4 et A5. Elles restent tout de même partagées avec ces lignes.

Nous verrons bien évidemment, à partir du chapitre 4, comment programmer et utiliser ces différentes ressources pour réaliser vos propres applications ou pour mettre en œuvre des *shields* du commerce.

Toutes ces entrées/sorties véhiculent des signaux numériques TTL, c'est-à-dire des signaux logiques ou signaux de type « tout ou rien » dont le niveau bas est nul et le niveau haut vaut 5 volts, mais l'Arduino sait aussi traiter les signaux analogiques.

■ Entrées analogiques

Il dispose pour cela de six entrées, repérées A0 à A5, qui peuvent admettre toute tension analogique comprise entre 0 et 5 volts (par défaut mais cela peut être modifié), mais pouvant bien sûr prendre n'importe quelle valeur dans cette plage puisque ce sont des entrées analogiques, contrairement aux signaux logiques évoqués précédemment.

Ces entrées analogiques sont gérées par un convertisseur analogique/numérique de 10 bits de résolution c'est-à-dire encore dont la sortie peut varier de 0 à 1023.

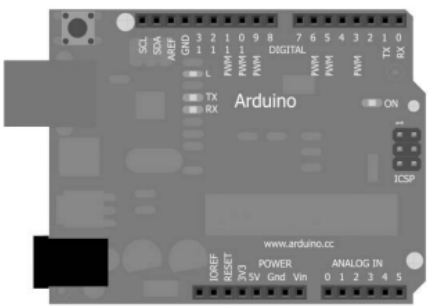

Figure 1.3 – Brochage des connecteurs des cartes Arduino Uno à partir de la Révision 3.

© Dunod – La photocopie non autorisée est un délit.

La figure 1.3 présente le brochage des connecteurs présents sur le pourtour des cartes Arduino ; connecteurs sur lesquels on retrouve bien sûr tous les signaux que nous venons de décrire.

> **Attention !**
> Sur les versions d'Arduino ayant existé avant la Révision 3 (Révisions 1, 2 et Arduino Duemilanove notamment) les broches SDA et SCL du connecteur à 8 points situé à côté du poussoir de Reset sont absentes. De même les deux broches situées à gauche de la broche Reset sur le connecteur repéré POWER sont absentes.

1.3.7 Différences entre Arduino Uno Révision 1, 2 et 3

Pour relier le microcontrôleur qui équipe les cartes Arduino à un micro-ordinateur *via* un port USB, il faut un circuit d'interface spécialisé. Sur les cartes Duemilanove et sur les premières Révisions des cartes Arduino Uno, cette tâche était confiée au circuit bien connu FT232 de FTDI, sur les cartes Uno Révision 2, c'est un Atmel ATmega 8U2 qui se charge de ce travail, tandis que sur les cartes Révision 3 c'est désormais un Atmel ATmega 16U2 qui gère cette interface USB.

Dans la majorité des cas, une telle différence passera totalement inaperçue tant pour tous les schémas, programmes et applications décrites dans ce livre que pour la majorité des applications que vous pourrez écrire. Vous n'aurez donc pas à vous soucier de cette différence.

Un autre point mérite d'être signalé car il peut avoir de l'importance selon l'usage que vous envisagerez de faire de votre Arduino. En effet, la carte Arduino Révision 3 existe, au moment où ces lignes sont écrites, en deux versions : une version avec le microcontrôleur ATmega 328 en boîtier DIL monté sur un support à 28 pattes classique, et une version avec l'ATmega 328 en boîtier CMS (composant à montage en surface) directement soudé sur la carte.

Ces deux versions sont évidemment 100 % équivalentes techniquement parlant mais, si vous souhaitez utiliser votre Arduino en tant que programmateur de microcontrôleur Atmel comme nous vous le proposons par exemple dans notre ouvrage *Arduino, Applications avancées* (même auteur, même éditeur) il faut impérativement utiliser une Arduino avec l'ATmega 328 en boîtier DIL car il est nécessaire de pouvoir le retirer de son support, ce qui est évidemment impossible avec la version CMS.

1.3.8 Schémas des Arduino Uno

Les Arduino étant des produits « open source » comme nous l'avons mentionné dans l'avant-propos, toutes les informations les concernant sont donc librement disponibles à titre d'information.

Les schémas des Arduino Uno ne font bien sûr pas exception à la règle et, même si vous pouvez parfaitement les ignorer dans la majorité des cas lors du développement de vos applications, ils sont à votre disposition sur le site officiel Arduino (*www.arduino.cc*) ou sur les compléments en ligne de cet ouvrage.

Si vous n'êtes pas électronicien et si vous trouvez ces schémas, disons… impressionnants, rappelez-vous que vous n'avez pas à en prendre connaissance. La seule chose qui va compter lors du développement de vos applications, ce sera les lignes d'entrées/sorties de l'Arduino que nous vous montrerons bien entendu comment utiliser dans la suite de cet ouvrage.

1.4 L'Arduino Mega 2560

La carte Arduino Mega 2560 a pris la relève de la carte Arduino Mega 1280 qui n'aura eu qu'une existence relativement éphémère. Elle ressemble à une carte Arduino Uno allongée comme vous pouvez le constater à l'examen de sa photo visible figure 1.4.

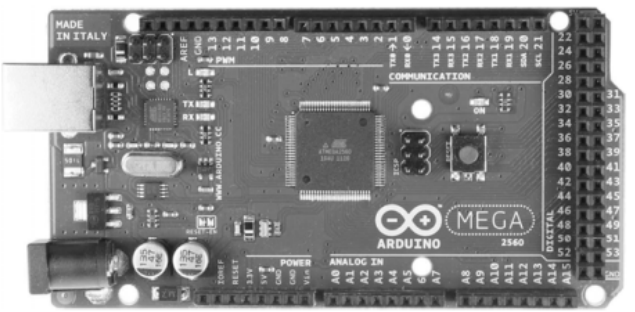

Figure 1.4 – L'Arduino Mega 2560 Révision 3 (doc. www.arduino.cc).

En fait, cette carte est pleinement compatible avec les Arduino Uno puisque les connecteurs situés sur sa partie gauche sont disposés de la même façon et véhiculent les mêmes signaux que sur ces dernières.

Même s'il existe des shields spécifiques de la Mega 2560, tous ceux destinés aux autres cartes Arduino Uno s'enfichent dessus et fonctionnent sans aucun problème.

La carte Mega 2560 est en effet elle aussi équipée d'un microcontrôleur Atmel AVR mais il s'agit cette fois-ci d'un ATmega 2560, d'où le nom donné à la carte ! Ce circuit peut être considéré comme une évolution de l'ATmega 328 qui équipe les Uno. Il comporte les mêmes ressources internes, comprend le même langage de programmation, mais dispose d'interfaces plus nombreuses.

Tout ce que nous avons écrit ci-dessus pour les cartes Arduino Uno pour ce qui est de l'alimentation, de l'horloge, de la liaison au PC ou au Mac et de la circuiterie de reset reste donc valable et nous ne le répéterons pas ici. Par contre, les mémoires et les entrées/sorties disponibles sont plus nombreuses ainsi que vous allez pouvoir le constater.

© Dunod – La photocopie non autorisée est un délit.

1.4.1 Les mémoires

L'ATmega 2560 qui équipe l'Arduino Mega 2560 dispose bien évidemment des trois mêmes types de mémoires que l'ATmega 328 des autres cartes Arduino à savoir : de la mémoire Flash pour le programme, de la mémoire vive ou RAM et de l'EEPROM. Tout ce que nous avons écrit à leur sujet au § 1.3.5 reste bien évidemment valable ; seules les tailles changent, dans le bon sens bien évidemment.

La mémoire Flash de programme voit ainsi sa capacité passer à 256 kilo-octets. La mémoire vive ou RAM atteint quant à elle les 8 kilo-octets tandis que la mémoire EEPROM se voit doter de 4 kilo-octets.

Le recours à l'Arduino Mega 2560 ne sera donc pas toujours imposé par un besoin en entrées/sorties plus nombreuses qu'avec l'Arduino Uno mais aussi parfois tout simplement par le besoin de tailles mémoires plus importantes.

1.4.2 Les entrées/sorties

On retrouve sur l'Arduino Mega 2560 les mêmes types d'entrées/sorties que sur l'Arduino Uno mais en nombre beaucoup plus important. Par ailleurs, du fait du plus grand nombre de pattes et donc d'entrées/sorties parallèles disponibles sur le boîtier de l'ATmega 2560, un certain nombre de ressources qui n'étaient pas directement accessibles avec l'ATmega 328 le deviennent ici. Voyons tout cela plus en détail.

■ Entrées/sorties numériques

Les quatorze lignes d'entrées/sorties numériques parallèles de l'Arduino Uno passent ici à cinquante-quatre, repérées de 0 à 53, mais gardent bien évidemment les mêmes caractéristiques que celles déjà vues lors de leur description précédente à laquelle nous vous renvoyons si nécessaire.

Ici aussi, certaines lignes partagent leur rôle avec certaines interfaces spécialisées contenues dans le microcontrôleur. Si ces interfaces ne sont pas utilisées par l'application, les entrées/sorties parallèles fonctionnent comme telles. Si ces interfaces sont utilisées par l'application, les lignes d'entrées/sorties parallèles qui sont partagées avec elles ne sont évidemment plus disponibles en tant que telles.

Sous ces conditions de partage on peut ainsi disposer de :

- quatre entrées (Rx0, Rx1, Rx2 et Rx3) et quatre sorties (Tx0, Tx1, Tx2 et Tx3) séries asynchrones, partagées respectivement avec les lignes 0, 19, 17, et 15 pour ce qui est des Rxn et 1, 18, 16 et 14 pour ce qui est des Txn ;
- six entrées d'interruptions externes partagées avec les lignes 2, 3, 18, 19, 20 et 21 ;
- quatorze sorties PWM (*Pulse Width Modulation* ou modulation de largeur d'impulsions) partagées avec les lignes 0 à 13 ;
- quatre entrées/sorties d'interface avec le bus série normalisé de type SPI réparties comme suit ; /SS sur 53, MOSI sur 51, MISO sur 50 et SCK sur 52.
- deux entrées/sorties d'interface série I2C réparties comme suit ; SDA sur 20 et SCL sur 21.

Ces entrées/sorties s'utilisent comme celles des Arduino Uno et tout ce que nous écrirons à leur sujet à partir du chapitre 4, sera bien évidemment valable aussi pour l'Arduino Mega 2560.

■ Entrées analogiques

Les entrées analogiques disponibles sur la carte Arduino Mega 2560 sont identiques dans leurs principes et possibilités à ce que nous avons vu ci-avant pour les Arduino Uno. Nous n'y reviendrons donc pas ici. La seule différence à signaler, mais elle est de taille, est que leur nombre passe de 6 à 16.

Du fait de ces nombreuses entrées/sorties supplémentaires, il a fallu doter l'Arduino Mega 2560 de connecteurs beaucoup plus nombreux ce qui explique l'allongement de la carte par rapport aux autres versions ; allongement qui a toutefois été conduit intelligemment puisque, répétons-le, les shields destinés aux Arduino Uno sont compatibles de l'Arduino Mega 2560 comme le confirme d'ailleurs le brochage des connecteurs de ce dernier visible figure 1.5.

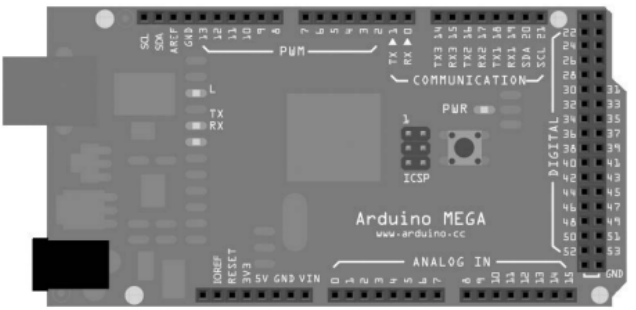

Figure 1.5 – Brochage des connecteurs de l'Arduino Mega 2560 Révision 3.

1.4.3 Schéma de l'Arduino 2560

Tout comme pour les Arduino Uno, le schéma de l'Arduino Mega 2560 est à votre disposition sur le site officiel Arduino (*www.arduino.cc*) ou sur les compléments en ligne de cet ouvrage.

Ce schéma étant encore plus impressionnant pour un non-électronicien que les précédents, rappelons que vous pouvez parfaitement l'ignorer puisque la seule chose qui va compter lors du développement de vos applications sera les lignes d'entrées/sorties de l'Arduino que vous découvrirez comment utiliser dans la suite de cet ouvrage.

© Dunod – La photocopie non autorisée est un délit.

1.5 Langage machine et langage évolué

Afin de réaliser une application à base de microcontrôleur, il faut disposer de plusieurs éléments essentiels. Le premier est évidemment la circuiterie minimum nécessaire à la mise en œuvre du microcontrôleur, telle que nous l'avons décrite au § 1.1. Dans le cas de l'Arduino, cette circuiterie est incluse sur la carte Arduino choisie, quelle qu'elle soit, et vous n'avez donc pas à vous en soucier.

Il faut ensuite réaliser la circuiterie d'interface entre le microcontrôleur et « le monde extérieur » c'est-à-dire les organes d'entrée et de sortie de l'application (clavier, afficheurs, relais, etc.). Dans le cas de l'Arduino, ces interfaces peuvent être disponibles prêtes à l'emploi dans le commerce ; ce sont les fameux shields dont nous avons déjà parlé et que vous découvrirez à partir du chapitre 3, ou elles peuvent être réalisées par vos soins, comme nous le verrons également dans la suite de cet ouvrage.

Le dernier élément, qui est bien souvent le plus important, est le programme qu'exécute le microcontrôleur pour faire fonctionner tout cela. Ce programme, qui peut être entièrement original ou qui peut faire appel à des portions de programme déjà écrites par d'autres utilisateurs, doit être écrit avec un programme spécial que l'on appelle l'environnement ou l'outil de développement.

Cet environnement de développement fonctionne, dans le cas de l'Arduino, sur n'importe quel PC, Mac ou ordinateur fonctionnant sous Linux. Il est donc compatible de tous les micro-ordinateurs que vous êtes susceptibles de rencontrer et d'utiliser. Nous allons voir ce qu'il comporte et préciser, à cette occasion, quelques notions importantes.

1.5.1 À l'origine : le langage machine

Le programme qu'exécute un microcontrôleur est constitué d'une suite d'instructions, codées en binaire, puisque ce sont là les seules informations que peut manipuler un tel circuit. Si, aux premiers âges de la micro-informatique, on était contraint de programmer en binaire, c'était tellement peu commode que très vite est apparu un langage, appelé langage machine, qui permet de représenter les instructions exécutées par le microcontrôleur par un « mot », plus compréhensible et surtout plus facile à mémoriser qu'un code binaire.

Ainsi par exemple, pour que le circuit réalise une addition, on n'est plus obligé d'écrire 11010011, code binaire de l'addition, mais tout simplement ADD, qui est le mnémonique de l'addition.

Un programme en langage machine est donc constitué d'une suite de mnémoniques correspondant aux instructions à exécuter, ce qui est plus lisible par le programmeur qu'une suite de codes binaires ! Mais, avant de pouvoir le placer en mémoire du microcontrôleur, cette suite de mnémoniques doit cependant être traduite en la suite de codes binaires correspondants, ce qui a lieu au moyen d'un programme spécial appelé l'assembleur.

Par déformation de langage, le programme constitué de la suite de mnémoniques, qui s'appelle normalement « programme en langage machine », a été baptisé

« programme assembleur » ou bien encore « programme en langage assembleur » ce qui n'est théoriquement pas correct puisque l'assembleur est en réalité, et comme nous venons de le dire, le programme qui traduit cette suite de mnémoniques en codes machines. Les erreurs ayant la vie dure, cette appellation incorrecte persiste encore aujourd'hui, même dans des ouvrages destinés à l'enseignement !

Ce langage machine présente cependant deux inconvénients majeurs. Le premier est qu'il est propre à chaque famille de microcontrôleurs. Ainsi, les AVR d'Atmel que l'on trouve sur l'Arduino ne parlent pas, par exemple, le même langage que les célèbres PIC de Microchip. Si donc vous voulez programmer différentes familles de microcontrôleurs, vous devez apprendre autant de langages machine que de familles.

Le deuxième inconvénient est que les instructions du langage machine sont tout à fait élémentaires. N'imaginez pas, par exemple, diviser deux nombres décimaux ou, pire encore, calculer une racine carrée. Le langage machine ne comporte en effet que des instructions très simples : addition, soustraction, plus rarement multiplication et quasiment jamais division mais sur de simples mots de 8 bits, c'est-à-dire compris, selon le cas, entre 0 et 255 ou entre –128 et +127. Ces instructions arithmétiques élémentaires sont toutefois complétées par des instructions logiques, des instructions de test et des instructions de saut ou d'appel de sous-programme. En d'autres termes, il faut très souvent de nombreuses instructions en langage machine pour réaliser une opération que vous jugeriez en apparence fort simple.

Malgré cela, de nombreux utilisateurs ont programmé ou programment encore en langage machine ; pas par masochisme mais parce que le langage machine présente, en contrepartie de ses inconvénients, deux avantages majeurs.

C'est un langage qui est au plus près du microcontrôleur puisque c'est directement celui qu'il comprend. Il permet donc de manipuler, avec un maximum de simplicité, ses ressources internes. En outre, et s'il est utilisé par un bon programmeur, c'est un langage qui permet de réaliser du code très compact et qui s'accommode donc bien des petites tailles des mémoires de programme disponibles dans les microcontrôleurs. N'oubliez pas, par exemple, que l'ATmega 328 qui équipe l'Arduino Uno ne contient que 32 kilo-octets de mémoire de programme !

Malgré cela, vous ne programmerez sans doute pas votre Arduino en langage machine mais en langage évolué dont nous allons voir dès à présent avantages et inconvénients.

1.5.2 Les langages évolués

La taille de la mémoire de programme des premiers microcontrôleurs était minuscule (parfois moins de 8 kilo-octets) et il n'était donc pas envisageable de les programmer en autre chose que le langage machine. Par contre, dès que cette taille s'est accrue, il est devenu possible d'envisager une programmation en langage évolué.

Qu'est ce qu'un langage évolué et quels sont ses avantages (et ses inconvénients) par rapport au langage machine nous direz-vous ? C'est assez simple à comprendre.

Un langage évolué, au rang desquels les plus connus sont aujourd'hui le Basic, le Pascal et le C, est un langage aussi proche que possible du langage « naturel », mais surtout un langage indépendant du microcontrôleur avec lequel il va être utilisé.

© Dunod – La photocopie non autorisée est un délit.

Avec un tel langage, même si vous changez de microcontrôleur, votre programme ne sera donc pas modifié ou très peu.

La raison d'être de cette particularité est facile à comprendre. Lorsque vous avez fini d'écrire un programme en langage évolué, et avant que celui-ci puisse être exécuté par le microcontrôleur que vous avez choisi, il doit être traduit en langage machine puisque c'est le seul langage que ce dernier comprend. Cette traduction est automatique grâce à un programme qui s'appelle un compilateur.

La chaîne de production complète d'un programme est ainsi la suivante lorsque l'on travaille en langage évolué. On commence par écrire le programme en langage évolué, ce qui constitue ce que l'on appelle le listing source ou, en raccourci, le source du programme. Ce source est fourni en entrée au compilateur ; compilateur qui doit bien sûr être adapté au microcontrôleur choisi. En sortie du compilateur, on dispose d'un programme en langage machine – les fameux mnémoniques qui ont été présentés précédemment – qu'il ne reste plus qu'à traduire en binaire avec le programme assembleur, également présenté ci-dessus.

Dans la majorité des cas, l'étape intermédiaire des mnémoniques et de l'assembleur n'est pas visible. On fournit donc, en entrée du compilateur, le source en langage évolué pour récupérer directement en sortie de celui-ci le code à programmer dans la mémoire du microcontrôleur. Ce code binaire, que l'on programme dans la mémoire du microcontrôleur, s'appelle l'objet du programme.

Ainsi, partant d'un « source » indépendant du microcontrôleur utilisé, on aboutit à un « objet » qui ne peut être utilisé qu'avec le microcontrôleur pour lequel le compilateur a été conçu.

Mais l'intérêt du langage évolué ne s'arrête pas là fort heureusement. Les instructions que comprend le langage évolué sont ainsi beaucoup plus puissantes que celles du langage machine. Les quatre opérations et de nombreuses fonctions scientifiques sont ainsi présentes et permettent de manipuler des nombres entiers mais aussi décimaux, en notations scientifiques et parfois très grands. Des instructions de prise de décision et de gestion de boucles faciles à comprendre sont également au rendez-vous.

Globalement, pour quelqu'un qui n'a jamais fait de programmation au préalable, l'apprentissage d'un langage évolué est en général considérablement plus simple que celle du langage machine.

1.5.3 L'environnement de développement de l'Arduino

L'Arduino n'étant ni plus ni moins qu'une carte à base de microcontrôleur, il peut théoriquement être programmé avec le langage de votre choix. Si, par exemple, vous maîtrisez le langage machine des microcontrôleurs de la famille AVR, vous pouvez donc parfaitement programmer votre Arduino en langage machine.

Cependant, l'approche adoptée par les promoteurs du projet ayant pour but de permettre à un maximum d'utilisateurs de développer leurs propres applications avec l'Arduino, un langage de programmation a été choisi et un environnement de développement complet, adapté à celui-ci, a donc été créé.

Cet environnement de développement est à ce jour disponible pour les PC fonctionnant sous Windows, sous Linux, et pour les Mac fonctionnant sous Mac OS X. Il comporte un éditeur de texte permettant d'écrire le programme source en langage évolué, un compilateur permettant de traduire ce programme source en programme objet ou, si vous préférez, en code binaire prêt à être placé en mémoire de l'Arduino, et un programmateur assurant justement la programmation de la mémoire de l'Arduino au travers du port USB dont il est équipé.

Cet environnement de développement ayant été conçu comme un tout, spécialement pour l'Arduino de surcroît, son utilisation est très confortable et ne requiert que très peu de manipulations pour passer de l'écriture d'un programme en langage évolué à son test en vraie grandeur sur l'Arduino, éventuellement muni de ses interfaces.

En effet, après avoir écrit le programme au moyen de l'éditeur intégré à l'environnement de développement, comme indiqué à titre d'exemple figure 1.6, il suffit d'un clic sur l'icône appropriée de sa barre d'outils pour qu'il se retrouve programmé dans la mémoire de l'Arduino et donc que ce dernier l'exécute aussitôt. On ne peut rêver plus simple.

Figure 1.6 – Fenêtre de travail de l'environnement de développement de l'Arduino.

© Dunod – La photocopie non autorisée est un délit.

Précisons avant de poursuivre que, tout comme les cartes d'interfaces ont été baptisées des shields, les programmes destinés à l'Arduino ont été appelés des sketches. Souhaitons que ces derniers vous fassent rire…

1.5.4 Le langage évolué de l'Arduino

On pourrait disserter pendant des heures sur les mérites comparés des différents langages de programmation évolués, et les informaticiens ne s'en privent d'ailleurs pas, mais ce n'est pas là notre propos. En effet, les développeurs du projet Arduino ont fait un choix et il faut donc « faire avec » si l'on souhaite utiliser l'environnement de développement fourni et, surtout, bénéficier de l'immense bibliothèque de logiciels déjà disponible sur Internet.

L'Arduino se programme donc en un langage qui est un mélange de C et de C++, restreint et adapté aux possibilités de la carte. L'intérêt de ce choix est que, même si le C et le C++ ne font pas partie des langages évolués les plus faciles à apprendre, ce sont tout de même aujourd'hui les plus enseignés et les plus utilisés en environnement professionnel.

Par ailleurs, comme il s'agit de C et de C++ restreints aux seules possibilités de l'Arduino, un certain nombre de fonctionnalités parfois difficiles à comprendre ne sont pas disponibles ce qui simplifie d'autant l'apprentissage du langage.

Vous découvrirez, au chapitre suivant, les bases de ce langage et ses règles de syntaxe mais vous verrez surtout, dans la suite de cet ouvrage et lorsque nous décrirons les cartes d'interface, comment l'utiliser au moyen des exemples de sketches que nous vous fournirons alors.

Sachez également que, compte tenu du succès planétaire de l'Arduino, plusieurs centaines de sketches sont disponibles sur Internet et que, sauf à développer une application très particulière, vous y trouverez très souvent votre bonheur, soit directement, soit après des adaptations minimes.

Et si, après avoir programmé l'Arduino avec « son » langage évolué, il vous prend l'envie de goûter aux joies du langage machine, sachez que nous avons consacré un livre aux microcontrôleurs AVR intitulé « Microcontrôleurs AVR : des ATtiny aux ATmega » chez Dunod bien entendu ; livre dont nous ne pouvons que vous recommander la lecture dans une telle situation !

1.6 L'environnement de développement

Afin de ne pas rester au stade des généralités et vous présenter un exemple concret d'installation de l'environnement de développement, nous avons choisi de travailler avec la version destinée aux PC fonctionnant sous Windows (de XP à Huit). L'installation des versions pour Linux ou Mac OS est similaire dans son principe et, si vous utilisez une de ces deux autres configurations, vous n'aurez donc aucune difficulté à la mener à bien en vous inspirant de la méthode utilisée pour la version Windows.

Dans tous les cas, une fois que l'environnement de développement est installé, son interface utilisateur est identique, que l'on soit sous Windows, Linux ou Mac OS. Quel que soit votre système de travail, vous n'aurez donc aucune difficulté à réaliser les exemples qui sont présentés dans la suite de cet ouvrage.

1.6.1 Installation de l'environnement de développement

La première étape consiste à télécharger la version du programme de l'environnement de développement correspondant à votre système d'exploitation. Même si de nombreux sites proposent aujourd'hui ce téléchargement, mieux vaut aller chercher le programme à sa source, c'est-à-dire sur le site officiel Arduino (*www.arduino.cc*), à la rubrique **Download**.

Choisissez la version Windows Installer, et non la version Windows ZIP file, qui sera beaucoup plus facile à installer car prenant automatiquement en charge la gestion des pilotes USB.

Une fois ce fichier d'un peu plus de 52 Mo téléchargé, lancez son exécution et, après avoir accepté son installation, laissez coché tout ce qui vous est proposé dans la fenêtre visible figure 1.7.

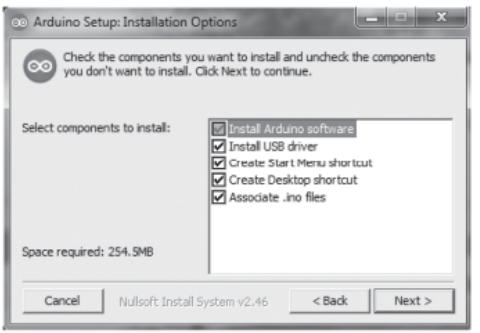

Figure 1.7 – La version Windows Installer installe tous les éléments nécessaires au bon fonctionnement de l'environnement.

Acceptez le répertoire d'installation proposé par défaut (C:\Program Files\Arduino) sauf si vous avez de sérieuses raisons de ne pas le faire, et laissez l'installation se poursuivre.

Après quelques dizaines de secondes, une fenêtre similaire à celle visible figure 1.8 vous demandera l'autorisation d'installer les pilotes USB. Avant de valider celle-ci en cliquant sur Installer, cochez la case située sur sa gauche afin de ne plus être importuné par ce type de question.

© Dunod – La photocopie non autorisée est un délit.

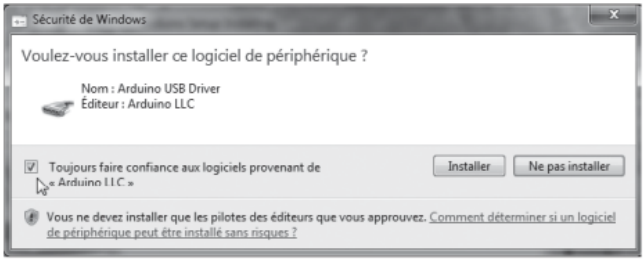

Figure 1.8 – Fenêtre de demande d'autorisation d'installation des pilotes.

L'installation se poursuit alors normalement et, lorsqu'elle est terminée, l'environnement de développement devient directement accessible *via* le raccourci qui a été placé sur votre bureau.

> **Attention !**
> De nombreux sites Internet, qui n'ont pas été mis à jour, proposent encore l'installation « à l'ancienne » c'est-à-dire à partir du fichier ZIP. Même si cette installation est, au final, équivalente à celle que nous venons de décrire, elle nécessite des manipulations fastidieuses au niveau de la gestion des pilotes USB ; manipulations qui n'ont plus lieu d'être aujourd'hui.

1.6.2 Première configuration de l'environnement de développement

Lors de la première utilisation de l'environnement de développement ou si, après avoir utilisé celui-ci avec une carte Arduino, vous décidez de changer de modèle de carte, il faut fournir à l'environnement deux informations distinctes : le type exact de carte Arduino utilisée d'une part et le port série virtuel utilisé par l'interface USB d'autre part.

Pour cela, lancez l'environnement et cliquez sur le menu **Outils**, sous-menu **Type de carte**. Dans la liste qui apparaît, cochez la ligne correspondant à votre version d'Arduino comme indiqué figure 1.9.

Ensuite, dans ce même menu **Outils**, sélectionnez le sous-menu **Port Série**. Cela aura pour effet de faire afficher une liste plus ou moins longue de ports COMx où x peut varier de 1 à 15 selon la configuration de votre PC. Sélectionnez un port COM libre qui deviendra celui affecté à l'Arduino.

> **Attention !**
> Selon ce qui a été utilisé et/ou installé préalablement à l'Arduino sur votre PC, il se peut que d'autres interfaces USB aient créé des ports COM virtuels. Si vous choisissez par erreur un de ces ports, la liaison avec l'Arduino ne pourra évidemment pas fonctionner et il conviendra donc de modifier votre choix.

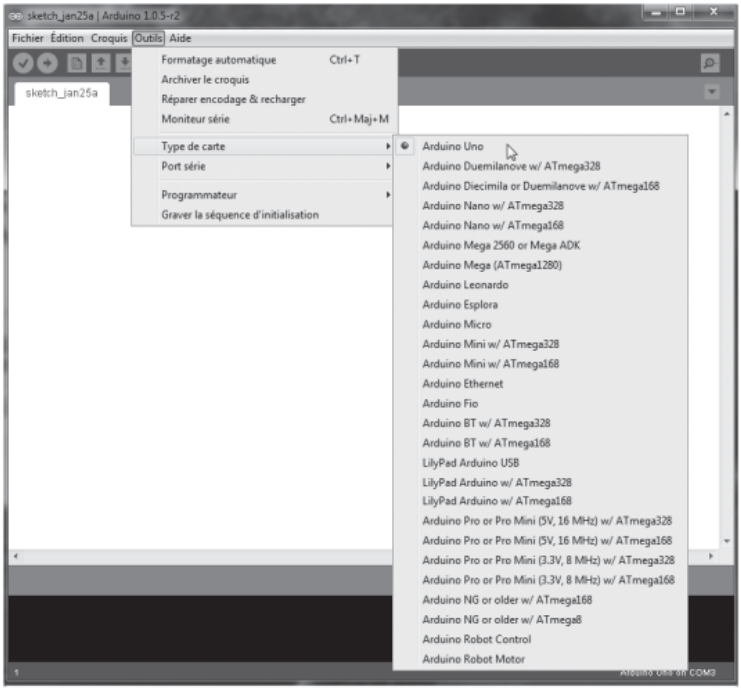

Figure 1.9 – Sélection du type de carte Arduino utilisé.

1.6.3 Utilisation de l'environnement de développement

La partie principale de la fenêtre de l'environnement de développement est occupée par la zone d'édition de programme, tandis que le haut de cette fenêtre comporte une barre de boutons permettant l'accès aux principales fonctions de l'environnement. En partie basse, on trouve une zone sur fond noir dans laquelle s'affichent les messages de service propres à certaines commandes, lors de du téléchargement du programme en mémoire de l'Arduino par exemple, ou bien encore les différents messages d'erreur de compilation.

L'éditeur proposé est classique et supporte, via le menu **Edition**, les commandes habituelles que sont couper, copier et coller ainsi qu'un certain nombre de fonctions plus spécifiques à l'écriture de programmes.

Commenter/Décommenter permet ainsi de transformer d'un seul coup une ligne ou un groupe de lignes de programme en commentaires ou vice versa. Cela s'avère très pratique lorsque l'on teste un programme et que l'on soupçonne une ou

© Dunod – La photocopie non autorisée est un délit.

plusieurs lignes de poser problème. Les transformer en commentaires permet ainsi de les désactiver sans avoir à les effacer du programme, et donc sans devoir les saisir à nouveau ensuite.

Augmenter l'indentation et **Réduire l'indentation** permettent quant à elles d'augmenter ou de diminuer l'indentation des lignes de programmes afin de rendre ces derniers plus lisibles et de bien distinguer des blocs d'instructions particuliers (contenus de boucles ou de zones d'exécution conditionnelle par exemple). C'est une sorte de commande de tabulation adaptée à l'écriture de programmes.

L'éditeur de texte dispose par ailleurs de la coloration syntaxique, certes assez simplifiée mais suffisante pour distinguer les instructions des noms de variables et des commentaires, ce qui aide beaucoup au début et permet de débusquer un certain nombre de fautes de frappe.

Signalons également la commande **Formatage automatique**, disponible dans le menu **Outils**, qui permet de formater automatiquement le programme en cours d'édition. Même si l'utilisation de cette commande, et donc le formatage d'un programme, n'est pas indispensable au bon fonctionnement de ce dernier, cela permet de le présenter conformément aux standards d'écriture du langage C et donc tout à la fois de le rendre plus lisible, mais aussi de permettre la mise en évidence d'erreurs telles que de mauvaises imbrications de boucles ou d'instructions de prise de décision par exemple.

En partie haute de cette fenêtre se trouve une barre de boutons qui couvrent 90 % des besoins lors de l'écriture et du test d'un programme. Leurs fonctions sont décrites dans le tableau 1.1.

Tableau 1.1 – Fonctions des icônes de la barre de boutons de l'environnement de développement.

Bouton	Fonction
	Vérifer : compile le sketch (programme) en cours d'édition et, de ce fait, vérifie sa syntaxe
	Nouveau : ouvre une fenêtre vierge pour écrire un nouveau sketch (programme)
	Ouvrir : ouvre la liste de tous les sketches contenus dans votre sketchbook. Le fait de cliquer sur un d'entre eux l'ouvre dans la fenêtre d'édition courante
	Enregistrer : sauvegarde le sketch en cours d'édition
	Téléverser : compile le programme en cours d'édition et le télécharge en mémoire de l'Arduino
	Moniteur série : ouvre l'écran du moniteur de port série intégré à l'environnement

À la lecture de ce tableau vous pouvez constater que l'on évoque la présence d'un moniteur de port série. L'environnement de développement intègre en effet un tel moniteur qui permet d'envoyer à l'Arduino et de recevoir de ce dernier des données via un port série du PC.
N'oubliez pas en effet que, même si votre Arduino est relié au PC via un port USB, celui-ci est vu par le PC comme un port série virtuel, celui que nous vous avons fait sélectionner au § 1.6.3, et qu'il est donc possible de l'utiliser pour communiquer avec l'Arduino au moyen d'instructions spécifiques prévues à cet effet.
Même si l'on peut utiliser pour cela l'hyper terminal de Windows, le moniteur de port série intégré à l'environnement de développement est nettement plus confortable et souple d'emploi.
D'autres commandes et fonctions sont à votre disposition mais il n'est pas indispensable de s'y intéresser à ce stade de l'ouvrage, d'autant que vous les découvrirez au fur et à mesure que vous écrirez des programmes.
Notez également que l'aide, accessible via le menu **Aide**, est très complète (mais en anglais !) et donne accès aux copies de la majorité des pages du site Internet officiel de l'Arduino (*www.arduino.cc*) ; pages qui sont stockées en local sur votre machine dans le répertoire d'installation de l'Arduino. Il n'est donc pas nécessaire d'être connecté à Internet pour utiliser cette fonction.

1.6.4 Un premier programme juste pour « le fun »

Arrivé à ce stade de la description de l'environnement de développement, nous vous proposons de le mettre en pratique avec un premier programme et si, lorsque l'on apprend la programmation classique on réalise traditionnellement un programme qui fait afficher « Hello World » sur un écran, avec un microcontrôleur on fait clignoter une LED.
Pour vous permettre de tester au plus vite l'environnement de développement et votre carte Arduino, toutes les versions passées et actuelles de ces cartes (Uno, Duemilanove et Mega 2560) intègrent une LED « de test » connectée entre la ligne de port parallèle 13 et la masse. Il n'est donc même pas nécessaire de câbler quoi que ce soit pour essayer votre premier programme ; programme qui est par ailleurs fourni prêt à l'emploi dans le répertoire d'installation de l'Arduino.
Après avoir lancé l'environnement de développement, cliquez sur le bouton **Ouvrir** (voir tableau 1.1 si nécessaire) et dans le menu **1.Basics**, sélectionnez **Blink**, ce qui a pour effet d'ouvrir le programme, visible figure 1.10, qui fait clignoter cette fameuse LED connectée à la ligne 13 du port parallèle.
Vous pouvez alors cliquer sur le bouton **Téléverser** pour lancer la compilation de ce programme suivie, si elle ne comporte pas d'erreur, de son téléchargement en mémoire de l'Arduino. Si tout se passe bien, ce qui devrait être le cas vu la provenance du programme (!), vous devriez voir apparaître, comme indiqué figure 1.10, le message **Téléversement terminé** en bas de la fenêtre d'édition ainsi que le message **Taille binaire du croquis : 1 084 octets (d'un maximum de 32 256 octets maximum)** dans la zone de messages de service. Cela signifie, d'une part que votre programme ne comportait pas d'erreur de compilation et qu'il a donc pu être chargé sans erreur en mémoire du microcontrôleur qui équipe l'Arduino et, d'autre

© Dunod – La photocopie non autorisée est un délit.

part, qu'il utilise 1 084 octets de sa mémoire de programme qui en comporte au maximum 32 256. On dispose ainsi d'une indication du taux d'occupation de cette dernière.

Figure 1.10 – Votre premier programme a été envoyé à l'Arduino.

Dès la fin du téléchargement du programme en mémoire de l'Arduino vous devez également observer le clignotement de la LED située à proximité du marquage 13 du connecteur de sortie numérique de l'Arduino. Si tel est le cas, vous êtes prêt à faire le grand saut…

2 • ENVIRONNEMENT DE DÉVELOPPEMENT ET LANGAGE DE PROGRAMMATION

Avec ce chapitre nous allons entrer de plain-pied dans la programmation de l'Arduino puisque nous allons décrire la syntaxe du langage de programmation. De ce fait, à ce stade de votre connaissance de l'Arduino, nous vous conseillons d'aborder la lecture de ce chapitre de la façon suivante :

– Si vous n'avez aucune notion particulière de la programmation en C ou en C++ vous n'avez pas besoin de lire l'intégralité de ce chapitre dès à présent. En effet, il est destiné à vous servir de référence une fois que vous aurez commencé à programmer l'Arduino car il présente toutes les instructions supportées, ainsi que leur syntaxe. De ce fait, sa lecture *ex abrupto*, à ce stade de l'ouvrage et si vous n'avez pas encore commencé à programmer l'Arduino, peut s'avérer quelque peu rébarbative (pour ne pas écrire « barbante » !).

  Nous vous conseillons donc d'installer l'environnement de développement, si ce n'est pas déjà fait suite à la lecture du chapitre précédent, et de lire dans un premier temps les informations relatives à l'organisation générale d'un programme et à sa syntaxe, puis de survoler la description des instructions avant de passer à la lecture des chapitres suivants.

  Ces chapitres expliquent comment réaliser et/ou utiliser des interfaces avec l'Arduino et présentent des exemples de programmes pour les mettre en œuvre. Lors de l'analyse de ces exemples, vous pourrez alors revenir lire, dans ce chapitre, la description détaillée des instructions qu'ils utilisent afin d'en comprendre le fonctionnement.

  Cette façon de procéder est, à notre avis, la meilleure pour apprendre la programmation de l'Arduino pour quelqu'un qui n'est pas familier de la programmation en C ou en C++.

– Si par contre le C ou le C++ n'ont plus de secrets pour vous, vous pouvez vous contenter de survoler ce chapitre afin de voir quelles sont les instructions qui sont à votre disposition et, surtout, quelles sont les fonctions spécifiques qui ont été ajoutées afin de gérer les entrées/sorties et les ressources du microcontrôleur qui équipe l'Arduino.

© Dunod – La photocopie non autorisée est un délit.

2.1 Organisation générale d'un programme

Le langage de programmation de l'Arduino étant dérivé du C et du C++, il respecte les règles de syntaxe propres à ces langages qui sont, au demeurant, relativement simples.

2.1.1 Instructions et commentaires

Une ligne qui commence par autre chose qu'un « slash » (/) ou un double « slash » (//) est considérée comme une ligne de code, ou d'instructions si vous préférez, et ne doit donc contenir que des informations compréhensibles par le compilateur.
Une ligne qui commence par un double « slash » (//) est considérée comme étant une ligne de commentaire. Son contenu est ignoré par le compilateur et vous pouvez donc y placer ce que bon vous semble.

```
// ceci est une ligne de commentaire
```

Comme sa longueur ne peut excéder une ligne, et pour éviter de devoir répéter les doubles « slashes » lorsque l'on veut faire plusieurs lignes de commentaires à la suite, il existe aussi une autre solution.
Ainsi, une ligne qui commence par un « slash » suivi par une astérisque marque le début d'une zone de commentaires ; zone qui se termine par une astérisque suivie d'un « slash ». Tout ce qui est compris entre ces deux marqueurs est ignoré par le compilateur.

```
/*
Ceci est une zone de commentaires
Elle peut comporter autant de lignes de commentaires
que vous le souhaitez
Elle se termine avec le marqueur ci-dessous
*/
```

Toute ligne de code, au sens large du terme, doit se terminer par un point-virgule.

> **Attention !**
> Le terme ligne de code doit être pris au sens large du terme. Ainsi une ligne de code vue par le compilateur ne correspond pas nécessairement à une ligne physique de l'éditeur. C'est uniquement le point-virgule qui marque la fin de la ligne de code. Les différents exemples ci-dessous sont ainsi tous valides.

```
// Ligne de code et ligne physique identiques
delay (1000) ;

// Une ligne de code répartie sur deux lignes physiques
if (x > 10)
digitalWrite (13, LOW) ;

// Deux lignes de code sur une même ligne physique
digitalWrite (13, HIGH) ; delay (1000) ;
```

Si les deux premiers exemples ci-dessus sont courants et quasiment imposés par les règles de syntaxe du langage, le dernier, à savoir deux ou plusieurs lignes de code sur une seule et même ligne physique est déconseillé car il contribue à rendre les programmes difficilement lisibles.
Les accolades quant à elles constituent le dernier élément important des règles de syntaxe du C et peuvent parfois poser problème lorsque l'on débute dans l'utilisation de ce langage. Elles ne peuvent s'utiliser que par paires et servent à délimiter un bloc d'instructions dans un certain nombre de circonstances particulières.
Ainsi par exemple, la définition d'une fonction y fait appel :

```
void loop () {
Instructions contenues dans la fonction loop ;
}
```

mais aussi la définition d'une boucle, quel que soit son type :

```
while (condition)
{
Instructions contenues dans la boucle ;
}
```

ou bien encore :

```
do
{
Instructions contenues dans la boucle ;
} while (condition) ;
```

Afin de vous aider dans la manipulation de ces accolades, l'éditeur de l'environnement de développement recherche automatiquement leur appariement. Ainsi, si vous positionnez le curseur juste après une accolade ouvrante, il encadre automatiquement l'accolade fermante associée. Cela permet ainsi très rapidement de déceler une éventuelle erreur d'appariement de ces dernières.

2.1.2 Inclusions et définitions

Afin de rendre un programme plus lisible, il est fréquent de remplacer certaines constantes par une appellation plus parlante que leur valeur numérique. Ainsi par exemple, si une LED est connectée à la patte 2 du microcontrôleur ; plutôt que d'indiquer 2 à chaque fois que l'on doit faire référence à la LED, il est plus logique et facile d'utiliser l'appellation LED. Pour ce faire on peut utiliser la directive `define` de la façon suivante :

```
#define nom de constante valeur
```

soit dans le cas de notre exemple :

```
#define LED 2
```

Attention !
Il ne faut en aucun cas terminer cette ligne par un point-virgule car elle n'est pas considérée comme une ligne de code. Il ne faut pas non plus faire figurer de signe égal entre le nom de la constante et sa valeur. Dans les deux cas, une erreur de compilation serait générée.

© Dunod – La photocopie non autorisée est un délit.

Dans le cas du langage C/C++ de l'Arduino, l'usage de `define` n'est toutefois pas conseillé et il est préférable d'utiliser la directive `const`. Cette directive réalise la même affectation que `define` mais permet, en outre, de préciser le type de donnée à utiliser (voir ci-après) ainsi que la visibilité de la constante ainsi définie (voir ci-dessous également). Elle s'utilise de la façon suivante :

```
const type de donnée nom de constante = valeur ;
```

soit, dans le cas de notre exemple précédent :

```
const byte LED = 2 ;
```

Attention !
Contrairement à `define`, il faut ici un point-virgule en fin de ligne et un signe égal entre le nom de la constante et sa valeur (qui a dit que les informaticiens étaient des gens logiques ?).

Lorsqu'un programme destiné à l'Arduino fait appel à des shields commerciaux ou déjà développés par d'autres utilisateurs, il est fréquent que les bibliothèques nécessaires aient déjà été écrites et soient disponibles sur Internet. Pour que ces bibliothèques puissent être utilisées dans votre propre programme ou sketch il faut les y inclure, ce qui a lieu tout naturellement avec la directive `include` qui s'utilise de la façon suivante :

```
#include <nom de la bibliothèque>
```

Ainsi par exemple pour utiliser la bibliothèque destinée à piloter un afficheur alphanumérique à cristaux liquides, on écrirait :

```
#include <LiquidCrystal.h>
```

Attention !
Tout comme `define`, la ligne contenant `include` ne se termine pas par un point-virgule.

2.1.3 Structure d'un programme

Un programme ou sketch destiné à l'Arduino, qui s'appelle désormais croquis suite à la francisation pas toujours judicieuse de l'outil de développement, comporte toujours deux fonctions distinctes. La première, appelée `setup`, n'est exécutée qu'une seule fois juste après le lancement du programme. Elle contient généralement des instructions d'initialisation de certaines ressources de la carte telles que, par exemple, définition des lignes des ports parallèles qui sont en entrées ou en sorties, définition de la vitesse de fonctionnement du port série, etc.
Elle s'écrit de la façon suivante :

```
void setup ()
{
Instructions à n'exécuter qu'une seule fois ;
}
```

La seconde fonction s'appelle `loop` (boucle en anglais) et contient les instructions qui seront indéfiniment répétées tant que l'Arduino restera sous tension. En d'autres termes, suite à un reset au moyen de son poussoir ou suite à une mise sous tension qui a pour effet de provoquer un reset automatique, l'Arduino exécute une seule fois les instructions contenues dans la fonction `setup` puis exécute ensuite indéfiniment les instructions contenues dans la fonction `loop` de ce même programme.
La fonction `loop` s'écrit de la façon suivante :

```
void loop ()
{
Instructions qui seront répétées indéfiniment ;
}
```

Selon le cas, une troisième partie peut ou non être présente mais ne contient pas d'instructions exécutables. Il s'agit de la zone de définition de constantes au moyen de `define` ou de `const`, ou bien encore d'inclusion de bibliothèques utilisées par le programme au moyen de `include`. Elle se place avant la fonction `setup`.
La structure générale d'un programme destiné à l'Arduino est ainsi la suivante :

```
définition éventuelle des constantes et bibliothèques

void setup ()
{
instructions à n'exécuter qu'une seule fois ;
}

void loop ()
{
instructions qui seront répétées indéfiniment ;
}
```

2.2 Données, variables et constantes

Quelle que soit la fonction d'un programme, il est toujours amené à manipuler des données dont la taille et le type peuvent être très différents ; d'un simple bit qui peut être égal à 0 ou à 1 s'il s'agit de données binaires destinées à allumer ou éteindre une LED par exemple, à un nombre comportant plusieurs chiffres si l'on réalise du calcul scientifique avec un tant soit peu de précision.

En outre, ces données ne sont pas seulement numériques mais peuvent aussi être alphanumériques afin, par exemple, de faire afficher des messages destinés à l'utilisateur de l'application.

Afin de ne pas gaspiller la précieuse mémoire du microcontrôleur qui est, rappelons-le, de petite taille, le compilateur doit être informé au mieux de la taille des données qu'il va être amené à manipuler afin de leur réserver en mémoire l'emplacement tout juste nécessaire à leur représentation. Ainsi, une donnée non signée inférieure ou égale à 255 n'occupera en mémoire qu'un octet tandis qu'un nombre en notation scientifique pourra utiliser quant à lui jusqu'à quatre octets.

© Dunod – La photocopie non autorisée est un délit.

Cette information du compilateur quant à la taille et aux types des données qu'il va devoir utiliser est impérative et doit être faite pour toutes les variables ou constantes utilisées dans un programme sous peine de générer une erreur lors de la compilation de ce dernier. Nous allons donc découvrir maintenant les différents types de données supportés par le compilateur de l'Arduino.

2.2.1 Les données numériques

Les données de types numériques sont au nombre de huit dont voici la description et les propriétés :

- **byte**

Une donnée de type `byte` est codée sur 8 bits et peut donc contenir un nombre non signé compris entre 0 et 255.

```
byte a = 145 ;     // la donnée a est égale à 145
byte a = B10010 ; // la donnée a est égale à 18
                   // (le B indique une notation en binaire)
```

- **int ou short**

Une donnée de type `int`, qui est le type le plus utilisé même si `byte` pourrait le remplacer très souvent, est codée sur deux octets, soit 16 bits, et peut donc contenir un nombre signé compris entre – 32 768 et + 32 767.

```
int b = -23456 ; // la donnée b est égale à -23456
```

- **unsigned int**

Comme son nom le laisse supposer, il s'agit de la version non signée de `int`. Toujours codée sur deux octets, une donnée de ce type peut donc être comprise entre 0 et 65 535 ($2^{16} - 1$).

```
unsigned int c = 45678 ; // la donnée c est égale à 45678
```

- **word**

Il s'agit d'une appellation qui fait double emploi avec `unsigned int`. Une donnée de ce type est codée sur deux octets et peut donc représenter un nombre compris entre 0 et 65 535.

```
word c = 45678 ; // la donnée c est égale à 45678
```

- **long**

Une donnée de type `long` est codée sur quatre octets, soit 32 bits, et peut représenter un nombre signé compris entre – 2 147 483 648 et + 2 147 483 647.

```
long d = 12345678 ; // la donnée d est égale à 12345678
```

■ unsigned long

C'est la version non signée de `long` qui occupe aussi quatre octets mais peut donc représenter un nombre compris entre 0 et 4 294 967 295 ($2^{32} - 1$).

```
unsigned long e = 4190857295 ; // la donnée e est égale à 4190857295
```

■ float

Une donnée de type `float` est ce que l'on appelle une donnée en virgule flottante ou bien encore, improprement, une donnée décimale. Elle occupe également quatre octets mais peut, grâce à la notation scientifique qu'elle utilise, représenter n'importe quel nombre compris entre $-3{,}4028235 \times 10^{38}$ et $+3{,}4028235 \times 10^{38}$. La précision offerte par l'Arduino est de 6 à 7 chiffres décimaux selon le cas.

Contrairement à ce que l'on rencontre avec d'autres compilateurs C, il n'est pas possible d'atteindre une précision de 15 chiffres en utilisant le type `double` qui, même s'il existe par soucis de cohérence du langage, est identique au type `float`. Cela n'est pas très contraignant car, généralement, on n'utilise pas un micro-contrôleur pour faire du calcul scientifique de haute précision.

```
float f = 1.245 ; // la donnée f est égale à 1,245
```

> **Attention !**
> Comme dans tout langage de programmation qui se respecte, « notre » virgule décimale est remplacée par un point.

■ double

Comme nous venons de le dire, ce type n'est là que par soucis de cohérence de définition du langage C. Dans le cas de l'Arduino, il est identique au type `float`.

2.2.2 Les données logiques et diverses

Les données logiques et diverses sont au nombre de deux seulement dont voici la description et les propriétés :

■ void

C'est une définition de donnée particulière, utilisée seulement lors de la description d'une fonction, pour indiquer que celle-ci ne fournit aucun résultat en retour à la fonction appelante. C'est ce mot clé que l'on retrouve ainsi, par exemple, lors de la définition de `setup` ou de `loop` comme nous l'avons vu au § 2.1.3.

```
void setup () // la fonction setup ne fournit aucun résultat
```

■ boolean

C'est une donnée logique qui ne peut contenir que deux valeurs `true` (vrai en anglais) ou `false` (faux). Elle occupe un octet en mémoire.

© Dunod – La photocopie non autorisée est un délit.

```
boolean marche = false ; // la variable marche est fausse
```

2.2.3 Les tableaux ou données indexées

■ **array**

Le type `array` désigne un tableau, que l'on peut aussi appeler donnée indexée. Il s'agit tout simplement d'une suite de données, regroupées sous le même nom, et dont chaque élément est repéré par son index. Ainsi par exemple si l'on veut repérer les pattes d'un circuit intégré on peut définir la donnée indexée `patte[]` qui contiendra par exemple 1, 4, 8 et 6 ; `patte[0]` sera égale à 1, `patte[1]` sera égale à 4 et ainsi de suite.

Un tableau pouvant contenir des données de n'importe quels types numériques ou alphanumériques, c'est le mot clé de définition du type choisi qui est employé pour définir le tableau sous l'une des formes suivantes :

```
int entrées[4] ;
byte pattes[] = { 1, 4, 8, 6} ;
char texte[8] = «bonjour» ;
```

Dans le premier cas on définit un tableau qui comprendra quatre données de type `int` appelées `entrées` ; données qui ne sont pas initialisées à une valeur particulière lors de cette déclaration.

Dans le deuxième cas, on définit un tableau contenant des données de type `byte`, appelées `pattes`, mais sans préciser le nombre d'éléments. Le compilateur comptera donc ceux contenus entre les accolades pour déterminer celui-ci. Notez que l'on réalise ici tout à la fois la définition du tableau et l'initialisation de son contenu.

Dans le troisième cas enfin, on définit un tableau un peu particulier puisque c'est une chaîne de caractères (voir ci-dessous). La taille du tableau doit, dans ce cas, être supérieure d'une unité au nombre de caractères de la chaîne car toutes les chaînes de caractères doivent être terminées par un caractère nul, non visible dans leur définition.

Pour accéder à n'importe quel élément d'un tableau, on utilise le nom du tableau et l'indice de la donnée désirée sachant que les indices démarrent à 0. Ainsi par exemple :

```
pattes [1] = = 4
texte [2] = = 'n'
```

De même, pour écrire une donnée dans un tableau, on utilise son indice de la façon suivante par exemple :

```
entrées [1] = = 4 ; // affecte la donnée 4 au 2ème élément de entrées
```

2.2.4 Les données caractères et chaînes de caractères

Les données de types caractères sont au nombre de quatre dont voici la description et les propriétés.

■ char

Ce type de donnée, quelque peu illogique dans son appellation, a théoriquement pour but de représenter n'importe quel caractère alphanumérique codé en ASCII. Il occupe un octet mais, comme il est signé, il peut représenter toute valeur numérique comprise entre – 128 et + 127. Utilisé pour représenter des caractères codés en ASCII, il réalise donc un certain gaspillage de mémoire puisque le code ASCII normal (non étendu) ne va que de 0 à + 127. Par contre, on peut aussi l'utiliser pour représenter une donnée numérique comprise entre – 128 et + 127. Notez par ailleurs qu'un caractère se représente entre deux guillemets simples.

```
char lettreA = 'A' // la donnée lettreA est égale au caractère A
                   // soit 65 en décimal qui est le code ASCII de A
char lettreA = 65  // est équivalent à l'exemple ci-dessus
                   // mais est moins lisible dans un programme
```

■ unsigned char

Ce type est évidemment la version non signée de `char`. Une donnée de ce type occupe donc toujours un octet et peut représenter tout nombre compris entre 0 et 255 ou bien encore un caractère en code ASCII étendu. Les propriétés de ce type sont identiques à celles de `byte` vu ci-dessus.

■ string

Ce type permet de représenter une chaîne de caractères mais, du fait que le langage de programmation de l'Arduino est un amalgame de C et de C++, deux formes différentes de représentation de chaînes sont aujourd'hui disponibles. La première, qui est aussi la plus simple d'emploi, se note `string` avec un s minuscule. La seconde, qui est ce que l'on appelle une classe en C++, se note `String` avec un S majuscule. Elle est d'un abord un peu moins naturel mais permet des manipulations de chaînes de caractères très puissantes, totalement inaccessibles à `string`. En contrepartie, elle utilise plus de place en mémoire, ce qui peut parfois être une entrave à son utilisation.

Intéressons-nous tout d'abord à `string` qui ne s'utilise pas sous forme de mot clé mais qui fait appel à une combinaison du type `char` et du type `array` comme nous l'avons déjà vu en partie ci-dessus avec la chaîne de caractères « bonjour ».

Une chaîne de caractère de type `string` peut être déclarée par l'une quelconque des méthodes ci-dessous :

```
char chaine1[10] ;
char chaine2[8] = { 'A', 'r', 'd', 'u', 'i', 'n', 'o'} ;
char chaine3[8] = { 'A', 'r', 'd', 'u', 'i', 'n', 'o', '\0'} ;
char chaine4[] = «Arduino» ;
char chaine5[8] = «Arduino» ;
char chaine6[10] = «Arduino» ;
```

© Dunod – La photocopie non autorisée est un délit.

Ces différentes déclarations ont les conséquences suivantes :

- `chaine1` comporte neuf caractères utiles. En effet, comme nous l'avons indiqué au § 2.2.3, une chaîne de caractères doit toujours être terminée par un caractère nul qui est bien évidemment pris en compte dans la définition de sa longueur totale.
- `chaine2` est égale à `Arduino` et comporte bien évidemment huit éléments qui sont les 7 lettres du mot Arduino et le caractère nul de fin.
- `chaine3` est évidemment identique à `chaine2` mais on a pris ici la peine de définir explicitement le caractère nul de fin (`'\0'`) ce que le compilateur n'impose nullement puisqu'il est capable de l'ajouter tout seul comme il l'a fait pour `chaine2`.
- `chaine4` est toujours égale à `Arduino` mais c'est au compilateur de compter les caractères pour déterminer sa longueur puisqu'elle n'est pas précisée entre les crochets de définition.
- `chaine5` est définie comme `chaine4` mais, comme nous avons fait le travail du compilateur, il ne faut pas commettre d'erreur sur la longueur et bien indiquer 8 entre crochets (les sept caractères de la chaîne et le caractère nul de fin).
- `chaine6` enfin est toujours égale à `Arduino` mais avec de la place vide en fin de chaîne puisque nous avons spécifié une longueur totale de dix caractères soit neuf caractères utiles. Il sera donc possible, si nécessaire, d'ajouter jusqu'à deux caractères à la chaîne Arduino.

■ array of strings

Comme son nom le laisse supposer, même si vous n'êtes pas anglophone, ce type de variable est un tableau de chaînes de caractères, c'est-à-dire encore un tableau à deux dimensions. Il fait appel à une notion qui effraye de nombreux apprentis programmeurs en C : les pointeurs. Dans notre cas, on pourra ignorer la définition exacte de ces pointeurs et les informations afférentes et se limiter à l'utilisation présentée à titre d'exemple ci-dessous.

Lorsque l'on fait appel à de tels tableaux de chaînes, c'est généralement parce que l'on doit utiliser plusieurs messages distincts, par exemple lors d'un dialogue avec l'utilisateur. On peut donc définir cet ensemble de chaînes contenant les messages de la façon suivante :

```
char* messages[] = {«message1», «message2», «message3», etc.} ;
```

Pour utiliser par exemple `message2` il suffira ensuite dans le programme de l'appeler de la façon suivante :

```
message[1]
```

N'oubliez pas en effet que les indices d'un tableau commencent à zéro. Le deuxième message de notre liste est donc bien `message[1]`.

> **Attention !**
> L'astérisque qui suit `char` n'est pas un effet de style mais matérialise, en langage C, le recours aux fameux pointeurs évoqués ci-dessus.

■ **String**

Le type `String` (avec un S majuscule) est ce que l'on appelle en C++ une classe. Cela désigne toujours bien sûr une chaîne de caractères mais à laquelle on peut appliquer un certain nombre de fonctions de manipulation de chaînes très évoluées (comparaison, longueur, découpage, remplacement de caractères, etc.).
Nous ne décrirons pas ces fonctions en détail à ce stade de l'ouvrage car, outre le fait que cela présenterait assez peu d'intérêt, leur connaissance n'est pas indispensable pour pouvoir utiliser l'Arduino et comprendre les différents exemples présentés dans la suite de cet ouvrage. Si vous désirez en savoir plus à leur sujet, nous vous renvoyons donc au guide de référence du langage (voir annexes).

2.2.5 Les conversions de types

Le langage C autorise la conversion ou le forçage d'un type de variable en un autre et le C de l'Arduino ne fait donc pas exception à cette règle. L'utilisation des fonctions de conversion est fort simple car ces fonctions portent tout simplement le même nom que le type dans lequel on souhaite convertir une variable.
Ainsi par exemple, pour convertir la variable `x` dans le type `byte` on écrira tout simplement :

```
byte(x) ;
```

Les fonctions de conversion disponibles sont donc tout simplement les suivantes : `byte(), char(), int(), word(), long()` et `float()`.
Seule la fonction `word()` présente une particularité qui est de pouvoir, si nécessaire, former un mot de 16 bits à partir de deux mots de 8 bits. Elle s'utilise pour cela de la façon suivante :

```
word (pforts, pfaibles) ;
```

Ceci a pour effet de former un mot de 16 bits dont les 8 bits de poids forts sont constitués par la variable `pforts` et les 8 bits de poids faibles par la variable `pfaibles`.

2.2.6 Visibilité des variables

Lorsque l'on définit une variable dans un programme en langage C sa visibilité (*scope* en anglais), c'est-à-dire les emplacements du programme qui pourront y faire appel, diffèrent selon la façon dont la variable a été définie.
Si l'on définit une variable au début du programme, avant la fonction `setup`, et donc *a fortiori* avant la fonction `loop`, cette variable sera considérée comme étant globale, c'est-à-dire qu'elle sera visible et utilisable dans tout le programme sans restriction.
Par contre, si l'on définit une variable au sein d'une fonction (`loop` ou `setup` par exemple) cette variable sera une variable dite locale et ne sera visible qu'au sein de cette fonction.

© Dunod – La photocopie non autorisée est un délit.

Enfin, il est également possible de définir une variable au sein d'une boucle de type `for` (voir ci-après), auquel cas la variable correspondante n'est visible que dans la boucle. Le petit exemple ci-dessous résume tout cela aussi bien qu'un long discours.

```
int varglobale ; // la variable varglobale est utilisable
                 // dans tout le programme

void setup ()
{
byte i ;         // cette variable i n'est utilisable
                 // que dans setup()
                 // son contenu est indépendant
                 // de celui de i définie dans loop()
instructions de la fonction setup ;
}

void loop()
{
int i ;          // cette variable i n'est utilisable
                 // que dans loop()
                 // son contenu est indépendant
                 // de celui de i définie dans setup()
                 // et de celui défini dans for

for (int i = 0 ; i<10 ; i++) {
                 // cette variable i ne concerne que la boucle for
                 // son contenu est indépendant de celui de i
                 // définie ci-dessus dans loop() et dans setup()
instructions de la boucle for ;
}
}
```

La visibilité des variables peut être partiellement modifiée au moyen de deux mots clés particuliers que sont `static` et `volatile` qui s'utilisent uniquement dans les situations suivantes.

■ static

Ce mot clé s'utilise lors de la définition des variables locales, c'est-à-dire des variables propres à une fonction. S'il n'est pas utilisé, le contenu de la ou des variables locales d'une fonction est réinitialisé à chaque appel de la fonction et est donc perdu d'une fois sur l'autre.

Par contre, si les variables locales sont définies en les faisant précéder du mot `static`, elles restent des variables locales, et ne sont donc toujours connues que dans la fonction où elles sont définies, mais leur contenu est préservé d'un appel de fonction à l'autre.

■ volatile

Ce mot clé ne doit être utilisé que lors de la définition des variables utilisées par les fonctions de gestion des interruptions (voir § 2.8.3).

2.2.7 Gestion de la taille des variables

Nous avons précisé précédemment quelle était la taille de chaque variable en fonction de son type mais il peut être utile, pendant le déroulement d'un programme, de disposer de cette information, plus particulièrement pour ce qui est des variables indicées ou tableaux. Il existe pour cela une fonction spécifique.

■ sizeof()

Cette fonction est capable de donner la taille de n'importe quel type de variable, c'est-à-dire en fait le nombre d'octets qu'elle occupe en mémoire. Elle s'utilise de la façon suivante :

```
taille = sizeof (variable) ;
```

`taille` contient alors le nombre d'octets utilisés en mémoire par la variable.

> **Attention !**
> Si la variable est une chaîne de caractères, il est normal que `sizeof` donne un résultat supérieur d'une unité au nombre de caractères de la chaîne puisque, comme nous l'avons vu ci-dessus, toutes les chaînes doivent être terminées par un caractère nul qui occupe donc un octet supplémentaire.

2.2.8 Constantes particulières

Un certain nombre de noms de constantes sont prédéfinis dans le langage C de l'Arduino et, même si les appellations choisies sont aisément compréhensibles, il est nécessaire d'apporter quelques précisions à leur sujet.

■ true et false

Nous avons déjà évoqué ces deux noms de constantes ci-dessus, lors de la présentation des données de type `boolean`, puisque `true` correspond à une donnée vraie alors que `false` correspond à une donnée fausse. En fait c'est un peu plus général que cela :

– `true` correspond à une donnée vraie, c'est-à-dire également à une donnée binaire qui est égale à 1 ou, plus généralement, à toute donnée numérique entière différente de 0.
– `false` correspond à une donnée fausse, c'est-à-dire à une donnée, binaire ou numérique peu importe, qui est nulle.

En d'autres termes, toute donnée numérique considérée comme donnée binaire est vraie (`true`) dès lors qu'elle est différente de 0 et fausse (`false`) dans le cas contraire.

■ HIGH et LOW

Comme leurs noms l'indiquent, ces deux constantes correspondent à des niveaux logiques respectivement haut et bas. Elles sont utilisées lorsque l'on veut manipuler les lignes d'entrées/sorties parallèles du microcontrôleur qui équipe l'Arduino et correspondent donc aux niveaux électriques présents sur ces lignes. Compte tenu

© Dunod – La photocopie non autorisée est un délit.

de la particularité des circuits logiques, ces niveaux sont différents selon que les lignes correspondantes sont utilisées en entrées ou en sorties.

Si une ligne de port est configurée en sortie, le fait de lui donner la valeur `HIGH` lui fait générer une tension proche de 5 volts, qui est la tension d'alimentation du microcontrôleur. Ceci n'est valable bien sûr que si la ligne correspondante n'est pas surchargée, c'est-à-dire si elle n'est pas obligée de débiter un courant supérieur au maximum permis qui est de 40 mA dans le cas des ATmega 328 et 2560.

Si une ligne de port est configurée en sortie, le fait de lui donner la valeur `LOW` lui fait générer une tension proche de la masse (0 volt). Comme pour `HIGH`, ceci n'est valable que si la ligne correspondante n'est pas surchargée c'est-à-dire si elle n'est pas obligée d'absorber un courant supérieur au maximum permis qui est aussi de 40 mA.

Si une ligne de port est configurée en entrée, le microcontrôleur y lira un niveau considéré comme `LOW` tant que la tension appliquée à cette entrée sera inférieure à 2 volts, et un niveau considéré comme `HIGH` si la tension appliquée est supérieure à 3 volts.

Et entre les deux nous direz-vous ? Et bien entre les deux c'est une situation qui ne devrait pas exister dans une application bien conçue. En effet, les normes relatives aux circuits logiques TTL stipulent que toute tension inférieure à 2 volts correspond à un niveau logique bas ou 0 et que toute tension supérieure à 3 volts correspond à un niveau logique haut ou 1 (en fait c'est un peu plus compliqué que cela mais on peut s'en contenter en première approximation). Entre ces deux tensions c'est la bande interdite, c'est-à-dire une zone dans laquelle le niveau logique n'est pas défini et peut donc être 0 ou 1 ou, pire, osciller entre 0 et 1 en fonction des perturbations diverses auxquelles peut être soumis le circuit. C'est donc une zone dans laquelle on ne rencontre pas de signal valide dans un circuit logique bien conçu.

■ INPUT, INPUT_PULLUP et OUTPUT

Ces trois dernières constantes particulières ne sont utilisées que dans la fonction `pinMode` qui précise le sens de fonctionnement des lignes d'entrées/sorties parallèles du microcontrôleur de la façon suivante :

```
pinMode (numéro de ligne, INPUT, INPUT_PULLUP ou OUTPUT) ;
```

Une ligne de port configurée en entrée par la constante `INPUT` est placée en mode haute impédance c'est-à-dire qu'elle ne consomme quasiment aucun courant sur l'élément qui lui est raccordé. Lorsqu'elle est lue par le programme, elle est au niveau `LOW` si la tension appliquée est inférieure à 2 volts et au niveau `HIGH` si cette tension est supérieure à 3 volts (voir ci-dessus).

Une ligne de port configurée en entrée par la constante `INPUT_PULLUP` se trouve reliée, à l'intérieur du microcontrôleur, par une résistance de tirage (de *pull up* en anglais) à son alimentation positive 5 volts. Si cette entrée est laissée « en l'air » c'est-à-dire si elle n'est reliée à rien, ou si elle est reliée à un circuit ouvert, l'entrée correspondante est alors vue comme étant au niveau haut du fait de cette résistance de tirage. Si tout cela vous semble pour l'instant incompréhensible, n'ayez aucune crainte, vous découvrirez ce qu'il en est au chapitre 4 où nous traitons en détail des entrées parallèles.

Une ligne de port configurée en sortie par la constante `OUTPUT` peut délivrer ou absorber un courant maximum de 40 mA. Dans cette limite, elle fournit une tension proche de 5 volts lorsqu'elle est placée au niveau `HIGH` et une tension proche de zéro (la masse) lorsqu'elle est placée au niveau `LOW`.

Attention !
Rappelons que, même si chaque sortie peut théoriquement fournir ou absorber jusqu'à 40 mA, le courant total fourni ou absorbé par l'ensemble des sorties ne saurait en aucun cas dépasser 200 mA.

2.3 Fonctions arithmétiques et mathématiques

Le langage de programmation de l'Arduino supporte bien entendu les traditionnelles quatre opérations arithmétiques ainsi qu'un certain nombre de fonctions mathématiques plus évoluées puisque même quelques fonctions trigonométriques sont présentes.

2.3.1 Opérateurs arithmétiques

■ **=**

Le premier opérateur est évidemment l'opérateur « égal » (=) ou opérateur d'assignation, à ne pas confondre avec l'opérateur « égal à » (= =) qui est un opérateur de comparaison que nous découvrirons ci-après même si, dans certains langages de programmation tels que le Basic par exemple, ces deux opérateurs sont confondus en un seul symbole.

L'opérateur égal sert donc à assigner une valeur à une constante ou à une variable. Ainsi `a = 2` signifie-t-il que l'on donne la valeur 2 à la variable a. Contrairement à l'égal dont on a l'habitude en algèbre, il n'est pas commutatif et `2 = a` conduirait à une erreur de compilation car cela n'aurait aucun sens. On ne peut pas en effet donner la valeur `a` au chiffre 2 !

■ **+, -, * et /**

Ces quatre opérateurs représentent respectivement l'addition, la soustraction, la multiplication et la division. Ils se comportent bien évidemment comme vous l'avez appris depuis l'école primaire mais avec tout de même quelques particularités liées aux types des variables utilisées lors de ces opérations. Ainsi par exemple, si l'on travaille avec des variables entières (types `byte` ou `int` par exemple) 7/3 sera égal à 2 car on réalisera alors une division entière !

Un certain nombre de règles ou de précautions sont donc à prendre en compte selon les types des variables utilisées :

– si les deux opérandes sont de types différents, le type pouvant contenir la variable la plus grande est utilisé pour le calcul ;
– de ce fait, si l'un au moins des opérandes est de type `float`, les opérations seront réalisées en virgule flottante ;

© Dunod – La photocopie non autorisée est un délit.

– il faut toujours choisir, pour la variable devant contenir le résultat, un type de taille suffisante sous peine de disposer d'un résultat faux ;
– un mauvais choix de type de variable peut conduire à un résultat faux (algébriquement parlant bien sûr). Ainsi par exemple, deux variables de types `int` égales respectivement à 50 et à 1 000 donneront un résultat négatif si on les multiplie car une variable de type `int` ne peut coder qu'un nombre compris entre – 32 768 et + 32 767 (revoir le § 2.2.1 si nécessaire).

Ces précisions étant vues, ces opérateurs s'utilisent de la façon suivante :

```
résultat = variable1 opérateur variable2 ;
```

■ **% ou modulo**

Cet opérateur permet tout simplement d'accéder au reste de la division entière. Il ne fonctionne donc qu'avec des variables entières c'est-à-dire avec tous les types de variables sauf `float` et `double` (qui est identique à `float` rappelons-le). Il s'utilise de la façon suivante :

```
résultat = dividende % diviseur ;
```

Ainsi par exemple, si `dividende` est égal à 9 et `diviseur` est égal à 4, `résultat` sera égal à 1 suite à l'écriture de :

```
résultat = 9 % 4 ;
```

puisque 9 divisé par 4 est égal à 2 avec un reste de 1.

2.3.2 Opérateurs arithmétiques composés

Comme leur nom l'indique, ces opérateurs n'ont aucune originalité en tant que tels mais résultent tout simplement de la composition des opérateurs précédents afin de rendre l'écriture des programmes plus compacte. Hormis les deux opérateurs d'incrémentation et de décrémentation qui sont d'usage courant et universel, nous vous déconseillons, au moins au début, de ne pas utiliser les autres car, lorsqu'on n'y est pas habitué, ils peuvent nuire à la lisibilité d'un programme.

■ **++ et - -**

Ces deux opérateurs sont utilisés pour incrémenter (augmenter) une variable d'une unité ou pour la décrémenter (diminuer) d'une unité, mais ils recèlent en outre une subtilité d'utilisation qui peut être la cause d'erreurs particulièrement difficiles à détecter.

– `i++` augmente ainsi i d'une unité tandis que `i - -` diminue i d'une unité mais
– `++i` augmente aussi i d'une unité tandis que `- -i` diminue aussi i d'une unité.

Dans le premier mode de notation, la variable *i* est évaluée puis est augmentée ou diminuée d'une unité. Dans le deuxième mode de notation, la variable *i* est augmentée ou diminuée d'une unité avant d'être évaluée. Ceci est évidemment très important lorsque l'on réalise des tests sur les valeurs de ces variables, lors de l'exécution de boucles par exemple (voir ci-après).

■ **+=, -=, *= et /=**

Comme nous l'avons laissé entendre ci-dessus, il ne s'agit là que d'artifices de notation qui rendent le code plus compact mais aussi, hélas, moins lisible. En utilisant ces opérateurs on peut ainsi remplacer :

```
x = x + 2 ;
```

par

```
x += 2 ;
```

et ainsi de suite avec les autres opérateurs composés.

2.3.3 Fonctions mathématiques

Outre ces quatre opérations de base, les fonctions mathématiques suivantes sont à votre disposition.

■ **min(x,y) et max(x,y)**

Ces deux fonctions s'utilisent et travaillent de la même façon mais, alors que `min` donne le minimum de deux variables, `max` donne le maximum. Elles s'utilisent tout simplement de la façon suivante :

```
minimum = min(x, y) ;  // minimum est égal à la plus petite
                       // des deux variables
maximum = max(x, y) ;  // maximum est égal à la plus grande
                       // des deux variables
```

■ **constrain (x, a, b)**

Cette fonction permet, comme son nom l'indique, de contraindre la variable x à rester comprise entre les deux bornes *a* et *b*. Elle est souvent utilisée pour limiter l'excursion de tension fournie, par exemple, par un capteur. Elle s'utilise tout simplement sous la forme :

```
contrainte = constrain (x, a, b) ;
           // la variable contrainte sera égale à
           // la variable x mais uniquement
           // tant que x sera comprise entre a et b
```

■ **abs(x)**

Il s'agit de la fonction mathématique classique valeur absolue ; `abs(x)` est donc égale à x si x est positive et à $-x$ si x est négative.

■ **map (valeur, debas, dehaut, versbas, vershaut)**

Cette fonction extrêmement pratique pour le traitement des informations délivrées par un capteur, permet de modifier, de façon homothétique, la plage de variation d'une variable. Les paramètres qu'elle utilise s'interprètent de la façon suivante.

© Dunod – La photocopie non autorisée est un délit.

La variable `valeur`, qui évolue normalement d'une valeur minimum appelée `debas`, à une valeur maximum appelée `dehaut`, évolue après application de cette fonction d'une valeur minimum appelée `versbas` à une valeur maximum appelée `vershaut`.
Cette fonction n'utilise que des variables entières mais sait gérer aussi bien les nombres négatifs que les nombres positifs.
On l'emploie très souvent avec la fonction de lecture d'une donnée analogique `analogRead` (voir ci-après) sous la forme suivante :

```
tension = map(analogRead(0), 0, 1023, 0, 500) ;
```

Le convertisseur analogique/digital de l'Arduino fournit en effet un résultat compris entre 0 et 1023 lorsque sa tension d'entrée évolue de 0 à sa valeur maximum (généralement 5 volts). Dans le cas de l'exemple précédent, on ramène cette variation de la plage 0-1023 à la plage 0-500. On peut ainsi ensuite faire afficher directement la variable `tension`, correspondant à la tension mesurée, exprimée en dizaine de millivolt (mV), sans nécessiter aucun calcul.

- **pow (base, exposant)**

Cette fonction permet d'élever la variable `base` à la puissance indiquée par la variable `exposant`. Le résultat obtenu est donc égal à $base^{exposant}$. Ceci n'a bien sûr de sens qu'avec des variables de type `float`, tant pour `base` que pour `exposant` et le résultat est obligatoirement de type `double`.

- **sqrt(x)**

Comme son nom abrégé le laisse supposer, cette fonction calcule la racine carrée de la variable *x*. Celle-ci peut être de n'importe quel type mais le résultat est obligatoirement de type `double`.

2.3.4 Fonctions trigonométriques

Trois fonctions trigonométriques sont proposées par le compilateur de l'Arduino : `sin(x)`, `cos(x)` et `tan(x)`. Elles calculent respectivement le sinus, le cosinus et la tangente de l'angle dont la valeur est contenue dans la variable *x*.
Cet angle est exprimé en radians et la variable *x* doit être de type `float` tandis que le résultat du calcul est obligatoirement de type `double`.

2.4 Opérateurs de comparaison et opérateurs logiques

Outre les opérateurs arithmétiques vus précédemment, le compilateur de l'Arduino dispose d'un certain nombre d'opérateurs de comparaison et d'opérateurs logiques hérités classiquement du langage C.

2.4.1 Opérateurs de comparaison

Les opérateurs de comparaison sont utilisés exclusivement avec la structure de contrôle de type `if` (voir ci-après) sous la forme :

```
if (condition)
{
   // exécution d'un certain nombre d'instructions
}
```

Si la condition est vraie, les instructions comprises entre les accolades sont exécutées. Dans le cas contraire, le programme se poursuit avec l'instruction qui suit l'accolade fermante.

Le tableau 2.1 présente les symboles et les fonctions des différents opérateurs de comparaison disponibles.

Tableau 2.1 – Opérateurs de comparaison.

Notation	Fonction
x = = *y*	*x* est égal à *y*
x ! = *x*	*x* est différent de *y*
x < *y*	*x* est inférieur à *y*
x > *y*	*x* est supérieur à *y*
x <= *y*	*x* est inférieur ou égal à *y*
x >= *y*	*x* est supérieur ou égal à *y*

Le court programme suivant a ainsi pour effet d'allumer une LED lorsque la variable `entrée` est strictement supérieure à 10.

```
if (entrée > 10)
{
digitalWrite(LED,  HIGH) ;
}
```

2.4.2 Opérateurs logiques booléens

Les opérateurs logiques, au sens réel de ce mot, sont, comme c'est le cas pour tous les langages de programmation, de deux types distincts. Les premiers sont généralement appelés opérateurs booléens et sont utilisés principalement pour combiner des expressions de comparaison. Ce sont eux que nous allons étudier en premier lieu.

© Dunod – La photocopie non autorisée est un délit.

■ **&& (et logique)**

Pour que la condition résultant de la combinaison de deux conditions au moyen d'un et logique soit vraie il faut que les deux conditions soient vraies. Ainsi par exemple :

```
if (a>2 && b<3)
```

ne sera satisfaite que si, simultanément, a est supérieur à 2 ET b est inférieur à 3.

■ **|| (ou logique)**

Pour que la condition résultant de la combinaison de deux conditions au moyen d'un ou logique soit vraie il faut que l'une au moins des deux conditions soit vraie. Ainsi par exemple :

```
if (a>2 || b<3)
```

sera satisfaite si a est supérieur à 2 ou si b est inférieur à 3. Comme il s'agit d'un OU logique inclusif, le fait que les deux conditions soient satisfaites est également valide.

■ **! (non logique)**

Cet opérateur a pour effet d'inverser l'état logique d'une variable. `!x` est fausse si `x` était vraie alors que `!x` est vraie si `x` était fausse.

2.4.3 Opérateurs logiques au niveau bit

Contrairement aux opérateurs précédents, et même s'il s'agit aussi d'opérateurs logiques qui portent de surcroît les mêmes noms que leurs prédécesseurs, on est ici en présence d'opérateurs agissant bit à bit sur les variables qu'ils manipulent.

Ils sont ainsi très utilisés pour réaliser des masques sur des données, pour éliminer certains bits, ou bien encore pour manipuler facilement des registres internes du microcontrôleur ou des circuits d'interface.

■ **& (et logique), | (ou logique), ^ (ou exclusif logique) et ~ (non logique)**

Ce sont les quatre fonctions bien connues mais, dans le cas où la logique ne serait pas votre tasse de thé, le tableau 2.2 rappelle leurs tables de vérité respectives.

Les trois premières fonctions s'utilisent de la même façon à savoir :

```
résultat = variable1 fonction variable2 ;
```

Quant à la fonction `not`, elle n'agit que sur une seule variable et s'utilise donc sous la forme :

```
résultat = ~variable ;
```

Elles peuvent agir sur tous les types de variables, sauf `float` et `double`, et réalisent toutes l'opération logique choisie bit à bit sur la ou les deux variables, selon le cas, conformément aux indications de la table de vérité du tableau 2.2. Ainsi par exemple :

```
byte a = B01101100 ;
byte b = B11000111 ;
resultat1 = a & b ;    // resultat1 vaut 01000100
resultat2 = a | b ;    // resultat2 vaut 11101111
resultat3 = a ^ b ;    // resultat3 vaut 10101011
resultat4 = ~a ;       // resultat4 vaut 10010011
```

Tableau 2.2 – Tables de vérité des fonctions logiques.

A	B	A & B (A ET B)	A | B (A OU B)	A ^ B (A OU exclusif B)	~A (NON A)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

En outre, les fonctions ET et OU supportent la combinaison avec le signe = comme nous l'avons vu au § 2.3.2 pour les opérateurs arithmétiques. On peut ainsi remplacer par exemple : `a = a & b` par `a &= b` et `a = a | b` par `a |= b` mais, comme nous l'avons indiqué ci-dessus, cela n'apporte rien et ne contribue pas à améliorer la lisibilité des programmes.
Deux autres fonctions logiques sont également disponibles mais sont parfois moins connues, puisque ce sont les décalages à droite ou à gauche, bit à bit, d'une variable.

■ << (décalage à gauche)

La fonction décalage à gauche décale d'un ou plusieurs bits vers la gauche la variable sur laquelle elle agit ; variable qui peut être de n'importe quel type sauf bien sûr `float` ou `double`. Les bits qui « sortent » de la variable par la gauche sont évidemment perdus tandis que ceux qui entrent dans la variable par la droite sont tous égaux à 0.
Cette fonction s'utilise de la façon suivante :

```
variable << nombre de bits ;
```

Ainsi par exemple :

```
byte a = B01101011 ;
byte b = a << 4 ; // b est égal à 10110000
```

Compte tenu des propriétés de l'arithmétique binaire, le fait de décaler une variable vers la gauche de n positions multiplie cette variable par 2^n. Hormis lors de ses utilisations purement logiques, cette fonction peut donc être aussi employée dans ce but.

© Dunod – La photocopie non autorisée est un délit.

■ **>> (décalage à droite)**

On pourrait croire que cette fonction est identique à la précédente, si ce n'est le sens du décalage, mais ce n'est pas tout à fait le cas en raison d'une particularité liée à la méthode de codage en binaire des nombres négatifs.

Comme son homologue, cette fonction décale donc bien d'un ou plusieurs bits vers la droite la variable sur laquelle elle agit ; variable qui peut être de n'importe quel type sauf bien sûr `float` ou `double`. Les bits qui « sortent » de la variable par la droite sont évidemment perdus, par contre ceux qui entrent par la gauche peuvent être égaux à 0 ou à 1.

Si le bit de poids fort de la variable avant son décalage était égal à 0, les bits qui entreront à gauche seront eux aussi égaux à 0. Par contre si ce bit de poids fort était à 1, les bits qui entreront par la gauche seront à 1. Cela tient au fait que, en binaire signé, le bit de poids fort d'une variable représente son signe (un 0 pour un nombre positif et un 1 pour un nombre négatif). Afin de conserver le signe lors des décalages, il convient donc d'appliquer la règle que nous venons de voir.

Ceci étant, cette fonction s'utilise comme son homologue :

```
variable >> nombre de bits ;
```

Ainsi par exemple :

```
byte a = B01101011 ;
byte b = B10110110 ;
byte c = a >> 4 ; // c est égal à 00000110
byte d = b >> 3 ; // d est égal à 11110110
```

Compte tenu des propriétés de l'arithmétique binaire, le fait de décaler une variable vers la droite de n positions divise cette variable par 2^n, mais ceci n'est valable que dans le cas où le bit de poids fort de la variable à décaler est égal à 0 (nombre positif donc). Hormis lors de ses utilisations purement logiques, cette fonction peut donc également être employée dans ce but.

2.5 Les structures de contrôle

Nous abordons là les éléments les plus importants du langage, puisque ce sont ces structures qui vont permettre des prises de décisions ou bien encore des réalisations de boucles en fonction de l'état de diverses variables. Leur syntaxe, et donc leur utilisation, est cependant fort simple comme nous allons le découvrir sans plus tarder.

2.5.1 Les prises de décision

Elles existent sous trois formes différentes et permettent de décider d'exécuter ou non une partie de programme en fonction de l'état d'une ou plusieurs variables. Nous allons donc les rencontrer sans cesse puisque, par exemple, le simple fait d'allumer une LED lorsque l'on actionne un poussoir y fait appel. Une telle action

peut en effet s'énoncer, en langage courant : si le poussoir est actionné, alors j'allume la LED. C'est exactement ce que vont faire ces structures de prise de décision.

■ if

La première d'entre elles, et la plus naturelle, est la fonction `if` qui s'utilise de la façon suivante :

```
if (condition)
{
instructions exécutées si la condition est vraie ;
}
poursuite de l'exécution ici si la condition est fausse ;
```

Ainsi par exemple, si l'on veut faire allumer une LED lorsqu'un poussoir est actionné, on pourra écrire :

```
if (poussoir = = 1)
{
digitalWrite (LED, HIGH) ;
}
```

Cette structure conditionnelle supporte quelques variantes de notation destinées à accroître la compacité d'un programme. On peut ainsi écrire également :

```
if (condition) instruction exécutée si la condition est vraie ;
```

ou bien encore :

```
if (condition) {instruction exécutée si la condition est vraie ;}
```

ou enfin :

```
if (condition)
instruction exécutée si la condition est vraie (1 seule ligne) ;
```

Par soucis de cohérence du langage et de lisibilité des programmes, nous vous conseillons néanmoins de respecter la notation initiale.
La condition peut faire appel à n'importe quel opérateur de comparaison vu au § 2.4.1 et, comme nous l'avons également précisé lors de la présentation des opérateurs logiques, il est possible de combiner plusieurs conditions comme dans l'exemple suivant :

```
if (poussoirA == 1 && poussoirB == 0)
{
digitalWrite (LED, HIGH) ;
}
```

Dans un tel cas, la LED ne s'allumera que si `poussoirA` est à 1 ET `poussoir B` est à 0.

© Dunod – La photocopie non autorisée est un délit.

■ if / else

Cette structure est une évolution de la précédente afin de permettre l'écriture plus facile d'une suite d'exécutions conditionnelles d'instructions. Sa syntaxe de base est la suivante :

```
if (condition)
   {
    instructions exécutées si la condition est vraie ;
   }
else
   {
    instructions exécutées si la condition est fausse ;
   }
```

Présentée sous cette forme, la combinaison `if/else` ne semble pas apporter grand-chose de plus que l'utilisation de `if` tout seul. Mais, où cela devient nettement plus intéressant, c'est lorsque l'on utilise l'autre forme de la combinaison `if/else` qui est la suivante :

```
if (condition 1)
   {
   instructions exécutées si la condition 1 est vraie ;
   }
else (condition 2)
   {
    instructions exécutées si la condition 1 est fausse ;
    et si la condition 2 est vraie ;
   }
else
   {
   instructions exécutées si la condition 1 est fausse ;
   et si la condition 2 est fausse ;
   }
```

Cette imbrication est nettement plus intéressante car elle permet de cascader des tests, d'autant qu'elle n'est pas limitée à deux conditions, comme indiqué dans l'exemple précédent, puisque le nombre de `else` suivis de conditions n'est pas limité.

Ceci étant, cette structure `if/else` est très pratique lorsque l'on n'a qu'un petit nombre de conditions à tester, comme dans l'exemple suivant qui mesure une tension analogique au moyen du convertisseur analogique/digital contenu dans le microcontrôleur de l'Arduino, et qui réalise trois actions différentes selon la plage où elle se trouve.

```
if (analogRead(0) < 100)
   {
   action exécutée si la tension est inférieure à 100 ;
   }
else (analogRead(0) > 500) ;
   {
   action exécutée si la tension est supérieure à 500 ;
   }
```

```
else
   {
   action exécutée si la tension est comprise entre 100 et 500 ;
   }
```

Cependant, et même si le compilateur est parfaitement capable de les gérer, la lisibilité et surtout l'analyse du comportement de votre programme risque de devenir difficile si vous imbriquez un trop grand nombre de `if/else` les uns dans les autres, et ce d'autant que, pour le cas où l'on doit faire de multiples tests, il existe une autre structure que voici.

■ switch / case

Cette structure permet de comparer, en une seule fois, la valeur d'une variable avec plusieurs valeurs distinctes et de choisir en conséquence les instructions qui seront alors exécutées. Elle s'utilise de la façon suivante :

```
switch (variable) {
  case valeur1 :
     instructions exécutées si variable = valeur1 ;
     break ;
  case valeur2 :
     instructions exécutées si variable = valeur2 ;
     break ;
  …
  case valeurn :
     instructions exécutées si variable = valeurn ;
     break ;
  default :
     instructions exécutées si variable n'est égal ;
     à aucune des valeurs précédentes ;
}
```

La variable est comparée successivement à `valeur1`, `valeur2`, etc. `valeurn` et, à la première égalité trouvée, les instructions comprises entre la ligne `case valeurx :` ayant détecté l'égalité de `variable` avec `valeurx` et la ligne `break` sont exécutées. Le programme continue ensuite son exécution après l'accolade finale.

Si aucune égalité n'est détectée, les instructions situées après `default :` sont exécutées. Notez toutefois que la présence de la section `default` est optionnelle.

L'instruction `break`, présentée par ailleurs ci-après, est optionnelle, mais généralement présente. Elle permet de terminer prématurément la structure `switch/case` après qu'une égalité ait été trouvée.

En son absence, ce qui est possible et utile dans certains cas, le comportement de `switch/case` devient alors le suivant. La variable est comparée successivement à `valeur1`, `valeur2`, etc. `valeurn` et, à la première égalité trouvée, les instructions placées après la ligne `case valeurx :` ayant détecté l'égalité de `variable` avec `valeurx` sont exécutées.

Contrairement à ce qui se passe en présence de `break`, on continue ensuite les comparaisons de la variable avec les valeurs suivantes présentes dans le `switch/case`

© Dunod – La photocopie non autorisée est un délit.

ce qui peut conduire à l'exécution de plusieurs des lignes de codes optionnelles qui y sont contenues.

2.5.2 Les boucles

Les boucles sont des éléments essentiels dans tout programme qui se respecte et le langage C de l'Arduino nous en propose donc trois types différents, susceptibles de couvrir la majorité des besoins.

■ for

La boucle `for`, qui n'est pas associée à `next` comme c'est le cas en Basic par exemple, est la plus connue et sans doute la plus utilisée en raison de sa très grande souplesse de définition. Sa syntaxe générale est la suivante :

```
for (initialisation ; condition ; incrément) {
   instructions exécutées dans boucle ;
}
```

Le premier élément, à savoir `initialisation`, n'est exécuté qu'une fois en début de boucle et définit tout à la fois la variable servant de compteur de tours de boucle et sa valeur initiale.
Le second élément, à savoir `condition`, s'applique au compteur de boucle et est testé avant chaque tour de boucle. Si la condition est réalisée, l'incrémentation qui suit a lieu et les instructions contenues dans la boucle sont exécutées. Si la condition est fausse, la boucle est ignorée et le programme poursuit son exécution après l'accolade finale.
Le troisième élément enfin indique comment est incrémenté (au sens large du terme car il peut diminuer !) le compteur de tours de boucle. Contrairement à ce que l'on rencontre dans d'autres langages de programmation tel que le Basic par exemple, cette incrémentation n'est pas limitée à des valeurs entières et les quatre opérations arithmétiques peuvent être utilisées pour faire évoluer la variable de boucle, même si cela reste assez rare.
Dans la majorité des cas, la syntaxe d'une telle boucle est donc fort simple comme dans l'exemple ci-dessous qui réalise l'augmentation progressive du rapport cyclique d'un signal PWM généré sur la sortie 2, en vue de graduer l'éclairage d'une ampoule par exemple.

```
for (byte i=0 ; i <= 255 ; i++) {
   analogWrite (2, i) ;
   delay (100) ;
}
```

Dans cet exemple le compteur de boucle est la variable `i`, de type `byte`, elle augmente d'une unité à chaque tour de boucle à cause du `i++` et la boucle se répète tant que `i` n'a pas atteint 256.

■ while

La boucle de type `while` est plus simple, mais aussi moins souple d'emploi que la boucle `for`. Elle s'utilise de la façon suivante :

```
while (expression) {
   instructions exécutées dans la boucle ;
}
```

Cette boucle s'exécute tant que l'expression, qui peut être n'importe quelle expression valide (variable, comparaison, etc.) est vraie. Par exemple, la boucle ci-dessous sera exécutée 10 fois :

```
byte i = 0 ;
while (i < 10) {
       instructions exécutées dans la boucle ;
       i++ ;
}
```

> **Attention !**
> L'expression est testée avant que ne soit exécutée la boucle. Si elle est fausse dès le début, la boucle ne sera donc jamais exécutée.

■ do / while

Cette boucle est une variante de la précédente pour laquelle l'expression n'est plus testée en début de boucle mais en fin. De ce fait, une boucle de ce type est toujours exécutée au moins une fois, quel que soit l'état de l'expression qui y est utilisée. Sa syntaxe est la suivante :

```
do
{
  instructions exécutées dans la boucle ;
} while (expression) ;
```

2.5.3 Les sauts

Les instructions de saut permettent généralement de sortir prématurément d'une boucle ou d'en court-circuiter une partie sous certaines conditions. Elles sont au nombre de quatre, même si l'utilisation de l'une d'entre elles est déconseillée sauf cas de nécessité absolue.

■ break

Ainsi que nous l'avons vu au § 2.5.1, cette instruction permet de quitter prématurément une structure de type `switch/case` mais elle peut aussi être utilisée dans une boucle `for, while` ou encore `do/while`.

Le fait d'exécuter `break` termine prématurément la structure dans laquelle elle est placée et fait poursuivre l'exécution du programme après l'accolade finale de cette

© Dunod – La photocopie non autorisée est un délit.

dernière. Voici un exemple d'utilisation, construit à partir du programme de gradation de l'éclairage d'une ampoule déjà vu précédemment.

```
for (byte i=0 ; i <= 255 ; i++) {
analogWrite (2, i) ;
delay (100) ;
if (poussoir = = 1)
       {
       i= 0 ;
       break ;
       }
}
```

Tant que la variable `poussoir` est égale à 0, la boucle se déroule normalement ; par contre, dès que celle-ci devient égale à 1, la condition testée par le `if` est vraie et l'instruction `break` est donc exécutée, ce qui fait terminer prématurément la boucle quel qu'en ait été son état.

■ continue

Cette instruction ne s'utilise qu'au sein d'une boucle de type `for, while` ou `do/while`. Elle permet de sauter une partie de la boucle mais sans interrompre cette dernière.

Si nous reprenons notre exemple de gradation précédent mais que nous voulons, par exemple, que les phases de gradation correspondant aux valeurs de `i` comprises entre 50 et 100 n'aient pas lieu on écrira tout simplement :

```
for (byte i=0 ; i <= 255 ; i++) {
  if (i > 50 && i < 100)
      {
      continue ;
      }
  analogWrite (2, i) ;
  delay (100) ;
}
```

Dès que `i` deviendra supérieur à 50 et tant qu'il restera inférieur à 100, la condition placée après le `if` sera vraie et l'instruction `continue` sera donc exécutée, ce qui aura pour effet de faire ignorer toute la fin de la boucle.

> **Attention !**
> Contrairement à `break` qui termine prématurément une boucle, `continue` ne fait ignorer que les instructions comprises entre elle-même et la fin de la boucle mais sans terminer prématurément cette dernière.

■ return

Cette instruction ne s'utilise que dans un cas particulier : à la fin de la définition d'une fonction, que ce soit pour la terminer, et donc assurer le retour à la fonction appelante, ou bien encore pour la terminer, assurer le retour à la fonction appelante

et, en même temps, transmettre un résultat à cette dernière. Deux notations sont ainsi possibles :

```
return ;
return valeur ;
```

Voici un exemple d'utilisation avec une fonction qui lit une grandeur analogique et qui fournit comme résultat 0 ou 1 selon que cette grandeur est supérieure ou non à 500.

```
int lectureanalogique () {
       if (analogRead(2) > 500)
             {
             return 1 ;
             }
       return 0 ;
}
```

■ **goto**

Cette instruction est honnie par de nombreux programmeurs puristes qui estiment avec dédain que, si elle est quasiment indispensable en Basic au point de rendre les listings confus et difficiles à suivre, elle est parfaitement inutile en C. Il convient de nuancer ces propos et, même s'il vaut mieux éviter d'écrire des `goto` à tout bout de champ, au point effectivement de rendre un programme difficile à suivre, le recours à cette instruction est parfois utile dans certaines situations.
Elle a pour effet de faire poursuivre l'exécution du programme à un emplacement repéré par une étiquette, et ce de manière inconditionnelle. La syntaxe est la suivante :

```
instructions diverses ;
goto labas ;
instructions qui ne seront pas exécutées ;
labas :
instructions qui seront exécutées suite au goto ;
```

2.6 Gestion du temps et des entrées/sorties

Avec ces fonctions, nous abordons des notions liées spécifiquement à l'Arduino puisque ces dernières n'existent pas en langage C standard. Elles ont donc été créées spécialement pour le compilateur C de l'Arduino afin de faciliter la gestion du temps et des entrées/sorties.

2.6.1 Gestion du temps

Quatre fonctions spécifiques ont été prévues pour gérer le temps au sein des programmes. Deux d'entre elles permettent de réaliser très facilement des boucles de temporisation alors que les deux autres sont d'usage moins courant.

© Dunod – La photocopie non autorisée est un délit.

■ delay()

Comme son nom le laisse supposer, cette fonction génère un certain temps de pause, ou délai, pendant lequel l'exécution du programme semble arrêtée (en fait le microcontrôleur ne s'arrête pas mais exécute le programme qui correspond à ce délai, programme qui ne fait rien d'autre que de consommer du temps). Le délai est ici exprimé en millisecondes au moyen de la variable utilisée, comme indiqué ci-dessous :

```
delay (variable) ;
```

Cette variable doit être du type `unsigned long` ce qui autorise la génération de délais pouvant être très longs. La ligne suivante génère ainsi, par exemple, une pause d'une seconde :

```
delay (1000) ;
```

■ delayMicroseconds()

Cette fonction est une variante de la précédente et s'utilise exactement de la même façon aux deux différences suivantes près : la variable utilisée est de type `unsigned int` et le délai est exprimé en microsecondes.
Ceci étant, et compte tenu de son mode de réalisation au niveau machine, l'exactitude de la durée du délai n'est assurée que si celui-ci est supérieur à 3 µs.

■ millis()

Cette fonction est un peu moins répandue que les deux précédentes dans le monde des compilateurs pour microcontrôleurs. Elle a pour but de donner comme résultat le temps qui s'est écoulé depuis le début de l'exécution du programme courant. Ce temps est exprimé en millisecondes et, du fait de son codage sous forme d'un `unsigned long`, il peut atteindre un maximum de 50 jours environ. Lorsque ce maximum est atteint, il revient à 0 mais le comptage se poursuit.

L'utilisation est fort simple comme indiqué ci-dessous :

```
unsigned long temps ;
temps = millis() ;
  // temps contient le temps écoulé en ms
  // depuis le début du programme
```

■ micros()

Cette fonction est très semblable à la précédente. Elle a pour but de donner comme résultat le temps qui s'est écoulé depuis le début de l'exécution du programme courant exprimé cette fois-ci en microsecondes. Du fait de son codage sous forme d'un `unsigned long`, il peut atteindre un maximum de 70 minutes environ. Lorsque ce maximum est atteint, il revient à 0 mais le comptage se poursuit.

L'utilisation est tout aussi simple que pour `millis` comme indiqué ci-dessous :

```
unsigned long temps ;
temps = micros() ;
// temps contient le temps écoulé en µs depuis le début du programme
```

Avec une carte Arduino fonctionnant à 16 MHz, la résolution de cette fonction est de 4 µs.

2.6.2 Entrées/sorties numériques

La gestion des entrées/sorties numériques est confiée à trois fonctions spécifiques que voici.

■ pinMode()

Comme nous l'avons vu au chapitre 1, les lignes d'entrées/sorties numériques des microcontrôleurs AVR qui équipent l'Arduino peuvent fonctionner indifféremment en entrées ou en sorties sous contrôle du programme. Il suffit pour cela de positionner correctement les bits de certains des registres internes qui les contrôlent dans le microcontrôleur.

La fonction `pinMode` se charge de cette tâche avec un maximum de confort en s'utilisant de la façon suivante :

```
pinMode (nom de la patte, sens de fonctionnement) ;
```

Le nom de la patte peut être exprimé directement, c'est-à-dire sous forme d'un chiffre compris entre 0 et 13 pour les Arduino Uno, ou entre 0 et 53 pour l'Arduino Mega 2560. Il peut aussi être exprimé sous forme d'un nom pour peu que l'on en ait affecté un au numéro de la patte concernée au préalable, comme indiqué ci-dessous.

```
byte LED = 13 ;
pinMode (LED, OUTPUT) ;
```

Le sens de fonctionnement ne peut quant à lui être égal qu'à trois valeurs : `INPUT`, `INPUT_PULLUP` ou `OUTPUT` comme nous l'avons vu précédemment, lors de la définition des constantes particulières.

Notez enfin que, comme les entrées analogiques peuvent aussi fonctionner en tant qu'entrées/sorties numériques, il est également possible de les faire figurer dans cette fonction. Elles s'appellent alors A0 à A5 pour les Arduino Uno et A0 à A15 pour l'Arduino Mega 2560.

■ digitalWrite()

Une fois que le sens de fonctionnement d'une ligne d'entrée/sortie numérique a été défini, il faut pouvoir la lire si elle est en entrée ou bien y écrire si elle est en sortie. C'est ce dernier rôle qui est tenu par `digitalWrite` qui s'utilise de la façon suivante :

```
digitalWrite (patte, valeur) ;
```

© Dunod – La photocopie non autorisée est un délit.

`Patte` est évidemment le numéro de la ligne d'entrée/sortie désirée ou le nom qui lui a été affecté comme indiqué ci-dessus, tandis que `valeur` ne peut être égale qu'à `HIGH` ou à `LOW` selon que l'on souhaite placer la ligne au niveau logique haut ou bas. Revoyez si nécessaires les explications que nous avons fournies ci-avant au § 2.2.8 sur les tensions correspondant à ces niveaux.

Pour faire clignoter une LED connectée entre la patte 13 de l'Arduino et la masse (celle qui est déjà câblée par défaut sur la carte) on pourra ainsi écrire par exemple :

```
byte LED = 13 ;
void setup()
{
pinMode (LED, OUTPUT) ;
}
void loop ()
{
     digitalWrite (LED, HIGH) ;
     delay (1000) ;
     digitalWrite (LED, LOW) ;
     delay (1000) ;
}
```

■ digitalRead()

Comme son nom le laisse supposer la fonction `digitalRead` assure la lecture de toute ligne d'entrée/sortie préalablement placée en entrée par `pinMode` et s'utilise de la façon suivante :

```
etat = digitalWrite (patte) ;
```

`Patte` est évidemment le numéro de la ligne d'entrée/sortie désirée, ou le nom qui lui a été affecté comme indiqué ci-dessus, tandis que `etat` se voit affecter la valeur `HIGH` ou `LOW` selon que la ligne est au niveau logique haut ou bas. Revoyez si nécessaires les explications que nous avons fournies ci-avant au § 2.2.8 sur les tensions correspondant à ces niveaux.

Pour faire allumer ou éteindre une LED connectée entre la patte 13 de l'Arduino et la masse (celle qui est déjà câblée par défaut sur la carte) au moyen d'un poussoir connecté entre la patte 12 et la masse par exemple, on peut ainsi écrire :

```
byte LED = 13 ;
byte Poussoir = 12 ;
byte etat = 0 ;
void setup()
{
      pinMode (LED, OUTPUT) ;
      pinMode (Poussoir, INPUT) ;
}
void loop ()
{
      etat = digitalRead (Poussoir) ;
      digitalWrite (LED, etat) ;
}
```

Attention !
Du fait de la particularité des entrées logiques, et sauf dans le cas où le mode `INPUT_PULLUP` est utilisé, une entrée non raccordée à quoi que ce soit peut prendre un niveau logique haut ou bas ou, pire, osciller sans cesse entre les deux. Il est donc normal dans un tel cas que `digitalRead` renvoie un résultat non significatif.

2.6.3 Entrées analogiques

Les microcontrôleurs AVR qui équipent les différents modules Arduino disposent tous d'un convertisseur analogique/digital dont le nombre d'entrées directement accessibles varie de 6 pour les Arduino Uno à 16 pour l'Arduino Mega 2560.

Ce convertisseur travaille sur 10 bits et la donnée numérique qu'il fournit en sortie est donc comprise entre 0 et 1023. Cette donnée est égale à 0 quand la tension d'entrée du convertisseur est nulle et égale à 1023 quand cette tension d'entrée est égale à une valeur particulière appelée tension de référence du convertisseur.

Par défaut, cette tension de référence est égale à la tension d'alimentation du microcontrôleur (5 volts ou 3,3 volts selon les modèles). Si donc la tension de référence est de 5 volts, le convertisseur délivrera 1023 lorsque sa tension d'entrée sera égale à 5 volts, ce qui lui conférera une résolution de 5/1024 soit environ 4,9 mV. Si par contre cette tension de référence n'est que de 2 volts, la résolution deviendra meilleure puisqu'elle atteindra alors 2/1024 soit environ 1,9 mV lorsque sa tension d'entrée sera de 2 volts.

Dernière précision à ce sujet : certains microcontrôleurs AVR intègrent une source de tension de référence interne qui peut être utilisée par le convertisseur analogique/digital.

Il est donc logique que le compilateur propose des fonctions permettant de gérer ce convertisseur. Ces fonctions sont au nombre de deux : une pour la lecture de la donnée convertie et une pour définir la tension de référence.

■ analogReference()

Cette fonction permet de préciser la tension de référence utilisée par le convertisseur analogique/digital. Elle s'utilise tout simplement de la façon suivante :

```
analogReference(type) ;
```

sachant que `type` ne peut prendre que l'une des valeurs suivantes :

- `DEFAULT` auquel cas la tension de référence utilisée est égale à la tension d'alimentation du microcontrôleur (5 volts sur les Arduino Uno et Mega 2560) ;
- `INTERNAL` auquel cas la tension de référence est égale à 1,1 volt sur les ATmega 328 qui équipent les Arduino Uno mais n'existe pas sur les Arduino Mega 2560 ;
- `INTERNAL1V1` auquel cas la tension de référence est égale à 1,1 volt mais uniquement sur les Arduino Mega 2560 ;
- `INTERNAL2V56` auquel cas la tension de référence est égale à 2,56 volts mais, ici encore, uniquement sur les Arduino Mega 2560 ;
- `EXTERNAL` auquel cas la tension de référence est égale à la tension appliquée sur l'entrée externe AREF de la carte Arduino.

© Dunod – La photocopie non autorisée est un délit.

Afin de ne pas détruire le convertisseur analogique/digital qui est contenu dans le microcontrôleur, veillez à bien respecter les trois recommandations suivantes :

– la tension d'entrée, appliquée sur n'importe quelle entrée analogique (A0 à AX), ne doit en aucun cas dépasser la tension d'alimentation du microcontrôleur ;
– si une tension de référence externe appliquée à AREF est utilisée, il faut impérativement le définir au moyen de `analogReference(EXTERNAL)` avant d'appeler la fonction `analogRead`. Dans le cas contraire, un court-circuit pourrait avoir lieu entre la source de tension de référence interne et cette tension externe pouvant conduire à la destruction du microcontrôleur ;
– enfin, et même si cela n'a pas de conséquences fâcheuses, la tension d'entrée appliquée au convertisseur doit toujours être inférieure ou égale à la tension de référence utilisée. Celui-ci délivre en effet en sortie la donnée maximum permise par ses 10 bits, à savoir 1023, lorsque la tension d'entrée est égale à la tension de référence. Toute valeur supérieure correspond donc aussi à 1023 mais cela n'a alors plus de signification.

■ analogRead()

Cette fonction permet bien évidemment de lire le résultat de la conversion de la tension présente sur l'entrée analogique sélectionnée. Elle s'utilise de la façon suivante :

```
tension = analogRead(entree) ;
```

où `entree` est le numéro de l'entrée analogique choisie (compris entre 0 et 5 ou 0 et 15 selon l'Arduino utilisé) ou le nom qui lui a été affecté. Le résultat fourni par cette fonction étant compris entre 0 et 1023 comme nous l'avons vu précédemment, la variable `tension` doit nécessairement être du type `int`.

Le convertisseur analogique/digital intégré au microcontrôleur qui équipe les cartes Arduino nécessite environ 100 µs pour réaliser une conversion complète. Il convient donc d'en tenir compte lors de l'utilisation d'`analogRead` et de se limiter à un maximum de 10 000 appels de cette fonction par seconde.

2.6.4 Sorties analogiques

Les microcontrôleurs qui équipent les cartes Arduino, tous modèles confondus, ne disposent d'aucun convertisseur digital/analogique au sens strict du terme. Il n'est donc pas possible de disposer directement, sur un des connecteurs de l'Arduino, d'une tension continue qui soit l'image d'une donnée numérique.

Par contre, ces microcontrôleurs sont capables de générer des signaux à modulation de largeur d'impulsion (MLI en français) appelés aussi signaux PWM pour *Pulse Width Modulation* en anglais

Ce sont des signaux logiques, rectangulaires, à une fréquence de l'ordre de 490 Hz dans le cas de l'Arduino, et dont le rapport cyclique est variable entre 0 et 100 %. Comme le montre la figure 2.1, rappelons que le rapport cyclique est le rapport

entre le temps pendant lequel le signal est au niveau et le temps pendant lequel il est au niveau bas, au sein d'une période.

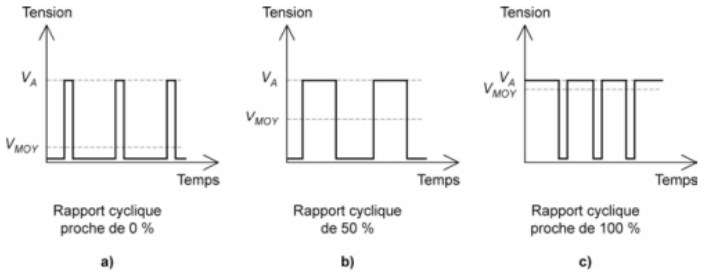

Figure 2.1 – Allure d'un signal PWM.

De ce fait, un signal de rapport cyclique nul est toujours au niveau logique bas tandis qu'un signal de rapport cyclique 100 % est toujours au niveau logique haut. Toutes les valeurs intermédiaires sont évidemment permises.

Même si ce signal n'est nullement une tension analogique, il peut le devenir si on lui fait subir un filtrage de type passe-bas même rudimentaire. Vous découvrirez tout cela au chapitre 5 dans lequel de tels signaux seront mis à profit ; nous n'allons donc pas traiter ce sujet ici car ce serait prématuré. Intéressons-nous donc seulement à la fonction prévue à cet effet par le compilateur C de l'Arduino.

■ analogWrite()

Contrairement à ce que pourrait faire penser son appellation, qui est un peu un abus de langage puisque les signaux PWM peuvent servir à autre chose qu'à générer des tensions analogiques, c'est bien cette fonction qu'il faut utiliser pour définir de tels signaux de la façon suivante :

```
analogWrite(sortie, rapportcyclique) ;
```

`sortie` est le numéro de la sortie PWM choisie (égal à 3, 5, 6, 9, 10 ou 11 sur les Arduino Uno et de 2 à 13 sur les Arduino Mega 2560), ou bien encore le nom qui lui a été affecté préalablement.

`rapportcyclique` est une variable de type `byte` (au minimum) contenant le rapport cyclique, compris ici entre 0 et 255. Un rapport cyclique de 0 correspond bien évidemment à 0 tandis qu'un rapport cyclique de 100 % correspond à 255.

Le signal PWM ainsi défini est généré, sur la sortie choisie et à la fréquence de 490 Hz environ rappelons-le, dès l'exécution de cette instruction et ce jusqu'à ce que l'un des phénomènes suivants se produise :

– définition d'un nouveau signal PWM sur la même sortie ;

– exécution d'une instruction `digitalRead` ou `digitalWrite` sur la même sortie.

© Dunod – La photocopie non autorisée est un délit.

Attention !
En raison de l'utilisation de la même ressource interne du microcontrôleur pour les fonctions `millis` et `delay` et pour la génération de signaux PWM sur les sorties 5 et 6, l'exactitude des rapports cycliques avec la valeur indiquée dans la fonction `analogWrite` n'est pas assurée dès lors que celui-ci est inférieur à 10. En particulier, un rapport cyclique de 0 peut ne pas conduire à la génération d'un signal en permanence au niveau bas sur ces deux sorties.

2.7 Entrées/sorties particulières

Afin de faciliter au maximum la mise en œuvre de certaines interfaces, quelques fonctions d'entrées/sorties particulières ont été ajoutées à celles que nous venons de voir. En voici la description.

■ tone()

Cette fonction fait générer un signal carré (rapport cyclique fixe de 50 %) basse fréquence (d'où son nom) sur la sortie choisie et pendant la durée définie. Pour ce faire, sa syntaxe respecte l'une des deux formes suivantes :

```
tone (sortie, fréquence) ;
tone (sortie, fréquence, durée) ;
```

– `sortie` est le numéro de la sortie choisie ou bien encore le nom qui lui a été affecté préalablement.
– `fréquence` définit la valeur de la fréquence choisie exprimée en Hertz (Hz).
– `durée`, dont la présence n'est pas obligatoire, définit la durée de génération de la note exprimée en milliseconde. Si cette durée n'est pas précisée, le signal est généré jusqu'à ce que soit appelée la fonction `noTone` (voir ci-dessous).

Pour pratique qu'elle soit, cette fonction est cependant affectée des restrictions suivantes :

– sur les cartes autres que l'Arduino Mega 2560, elle perturbe les sorties PWM 3 et 11 ;
– il n'est possible de générer qu'une seule tonalité à un instant donné et, si une note est déjà en cours de génération, l'appel de la fonction `tone` pour une autre sortie est alors sans effet ;
– si une note est en cours de génération sur une sortie, l'appel de la fonction `tone` pour cette même sortie permet de modifier sa fréquence à la volée ;
– pour jouer une mélodie sur différentes sorties, il est nécessaire de faire appel à la fonction `noTone` à chaque changement de sortie.

■ noTone()

Comme indiqué précédemment, cette fonction permet de faire cesser la génération d'une tonalité sur une sortie déterminée. Elle s'utilise tout naturellement de la façon suivante :

```
noTone (sortie) ;
```

L'appel de cette fonction alors qu'aucune tonalité n'est en cours de génération est sans effet mais ne provoque aucune erreur.

■ pulseIn()

Comme son nom le laisse supposer, même pour un non anglophone, cette fonction permet de mesurer la durée d'une impulsion. Elle s'utilise sous l'une des deux formes suivantes :

```
durée = pulseIn(entrée, type) ;
durée = pulseIn(entrée, type, délaimax) ;
```

- `entrée` permet évidemment de définir l'entrée utilisée, soit sous forme numérique soit, comme nous l'avons déjà vu de multiples fois, au moyen d'un nom préalablement défini.
- `type` permet de définir le type d'impulsion. Il peut être égal pour cela à `HIGH` ou à `LOW`. S'il est égal à `HIGH`, la fonction attendra que l'entrée passe du niveau bas au niveau haut et mesurera le temps pendant lequel elle restera à ce niveau. S'il est égal à `LOW`, ce sera bien évidemment le contraire qui se produira.
- `délaimax` est une variable de type `unsigned long` qui précise le délai d'attente maximum de changement d'état de l'entrée. Ce délai est exprimé en microseconde (µs). Si `délaimax` n'est pas précisé, sa valeur par défaut est d'une seconde. Si le premier front de l'impulsion ne se produit pas avant `délaimax` (qu'il soit explicitement précisé ou qu'il soit égal à sa valeur par défaut) la fonction `pulseIn` retourne un résultat nul.
- `durée` est évidemment la durée de l'impulsion et doit être du type `unsigned long`.

Compte tenu du principe de programmation de la fonction `pulseIn`, le résultat fourni ne peut être considéré comme correct que dans la plage allant de 10 µs à 3 minutes.

■ shiftOut()

Cette fonction permet de faire générer une donnée sous forme série synchrone sur une quelconque des sorties du microcontrôleur qui équipe l'Arduino. Elle est très utile pour communiquer avec de nombreux circuits externes tels que des mémoires EEPROM, des contrôleurs d'affichage ou même de simples registres à décalage. S'agissant d'une sortie de type série synchrone, cette fonction génère deux signaux distincts : celui qui véhicule les données d'une part et celui qui véhicule l'horloge, permettant de prendre correctement en compte ces données, d'autre part.

Même si cette fonction est utilisable avec des circuits externes disposant d'une interface de type SPI, sachez qu'il existe une bibliothèque spécifique SPI dans le volumineux catalogue de bibliothèques de l'Arduino et qu'il vaut mieux, dans un tel cas, y faire appel plutôt que d'utiliser `shiftOut`.

Si ces notions ne vous sont pas familières, sachez que vous découvrirez, au chapitre 6, divers exemples d'utilisation tant de cette fonction que de la bibliothèque SPI.

© Dunod – La photocopie non autorisée est un délit.

Ceci étant précisé, la syntaxe de `shiftOut` est la suivante :

- `shiftOut (sortie donnée, sortie horloge, ordre des bits, donnée) ;`
- `sortie donnée` permet évidemment de définir la sortie utilisée pour les données sous forme série, soit de façon numérique soit au moyen d'un nom préalablement défini.
- `sortie horloge` permet quant à elle de définir la sortie utilisée pour le signal d'horloge, soit de façon numérique soit au moyen d'un nom préalablement défini.
- `ordre des bits` qui peut être égal seulement à `MSBFIRST` ou à `LSBFIRST` permet de définir dans quel ordre vont sortir les différents bits qui composent la donnée sur la ligne qui leur est affectée. `MSBFIRST` fait ainsi sortir la donnée avec le bit de poids fort en premier tandis que `LSBFIRST` fait sortir la donnée avec le bit de poids faible en premier.
- `donnée`, qui est une variable de type `byte`, est la donnée à faire sortir sous forme série.

Comme indiqué précédemment, nous vous renvoyons au chapitre 6 pour un exemple concret d'utilisation de cette fonction.

2.8 Fonctions diverses

Nous avons regroupé sous ce vocable les fonctions qui n'ont pu être placées dans la classification précédente. Elles n'en sont pas moins utiles pour autant comme nous vous proposons de le découvrir sans plus tarder.

2.8.1 Génération de nombres aléatoires

La génération de nombres aléatoires est fréquemment utilisée en programmation mais, paradoxalement, il peut s'avérer difficile de disposer de nombres qui soient vraiment aléatoires si on ne prend pas un minimum de précautions. Pour ce faire, le langage de l'Arduino dispose donc de deux fonctions distinctes.

■ randomSeed()

En informatique, un nombre aléatoire est toujours généré à partir d'une valeur de départ que l'on appelle en anglais le *seed*, ce qui signifie en français la semence. Si l'on part du même *seed*, on générera toujours la même séquence de nombres aléatoires.

En d'autres termes, les nombres de cette séquence seront bien aléatoires, les uns par rapport aux autres si l'on peut dire mais, si l'on fait générer deux fois de suite cette séquence avec le même *seed* de départ, on obtiendra la même suite de nombres aléatoires à chaque fois. Si l'on réalise par exemple un jeu de dés électronique de cette façon, cela peut être fâcheux (sauf pour les tricheurs !).

La fonction `randomSeed` permet donc d'initialiser ce *seed*, soit à une valeur fixe auquel cas les appels successifs à la fonction `random` feront générer la même suite de nombres aléatoire, soit à une valeur quelconque, auquel cas les appels successifs à la fonction `random` feront générer des suites différentes de nombres aléatoires.
La syntaxe de cette fonction est la suivante :

```
randomSeed(valeur) ;
```

`valeur` est la donnée de type `int` ou `long` qui initialise le `seed`.
Comme dans la majorité des cas où on utilise des nombres aléatoires, on souhaite que ces derniers le soient vraiment, il faut trouver un moyen pour que `valeur` soit différente à chaque exécution du programme. La solution la plus simple pour cela consiste à utiliser un évènement extérieur tel que, par exemple, la mesure d'une tension analogique sur une entrée non connectée.
En effet, une telle entrée est sensible aux perturbations électromagnétiques avoisinantes et la tension qui y est présente varie donc constamment. Même si c'est dans de faibles proportions, cela suffit à faire générer des nombres différents à la fonction `analogRead` compte tenu de sa résolution.
La ligne suivante :

```
randomSeed(analogRead(0)) ;
```

permet alors d'initialiser le seed avec une valeur vraiment aléatoire lors de chaque exécution du programme.

■ random()

Cette fonction permet de générer des nombres aléatoires calculés à partir du seed initialisé par la fonction `randomSeed`. Elle peut s'utiliser sous les deux formes suivantes :

```
nombre = random (max) ;
nombre = random (min, max) ;
```

`nombre`, `min` et `max` sont tous de type `long`.
La première forme permet de disposer d'un nombre aléatoire compris entre 0 et `max-1` tandis que la seconde forme permet de disposer d'un nombre aléatoire compris entre `min` et `max-1`.

2.8.2 Manipulation de bits

Même s'il est possible de manipuler des bits au sein de variables en utilisant les fonctions logiques vues au § 2.4.3 cela peut vite devenir lourd et fastidieux. C'est pourtant une opération indispensable dans bien des cas lorsque l'on veut agir sur les registres internes du microcontrôleur ou bien encore sur des registres contenus dans des circuits externes, pour les paramétrer ou les initialiser par exemple. Un certain nombre de fonctions, décrites ci-dessous, ont donc été prévues pour simplifier la majorité de ces manipulations.

© Dunod – La photocopie non autorisée est un délit.

■ **lowByte()**

Cette fonction extrait les 8 bits de poids faibles d'une variable de n'importe quelle taille. Elle s'utilise de la façon suivante :

```
resultat = lowByte(variable) ;
```

`resultat` est évidemment du type `byte` tandis que `variable` est de n'importe quel type.

■ **highByte()**

Contrairement à ce que son nom laisse supposer, cette fonction n'est pas tout à fait le contraire de la précédente. En effet, elle extrait le deuxième octet en partant de la droite d'une variable de n'importe quelle taille. Si cette variable est de type `word`, ce sera effectivement l'octet de poids fort (puisqu'il n'y a que deux octets dans un mot de 16 bits). Si cette variable est de taille plus importante (`long` par exemple) cette fonction n'extraira toujours que le deuxième octet en partant de la droite qui ne sera donc pas l'octet de poids le plus fort.
Elle s'utilise de la façon suivante :

```
resultat = highByte(variable) ;
```

`resultat` est évidemment du type `byte` tandis que `variable` est de n'importe quel type.

■ **bitRead()**

Cette fonction permet de lire directement l'état d'un bit d'une variable sans avoir besoin de recourir aux fonctions logiques vues précédemment. Elle s'utilise sous la forme :

```
resultat = bitRead(variable, n) ;
```

`n` permet de spécifier le numéro du bit à lire sachant que celui-ci commence à 0 pour le bit de poids le plus faible, c'est-à-dire le bit le plus à droite de la variable.

■ **bitWrite()**

Cette fonction est le complément de la précédente et permet d'écrire directement un bit dans une variable sans avoir besoin de recourir aux fonctions logiques vues précédemment. Elle s'utilise sous la forme :

```
bitWrite (variable, n, bit) ;
```

`n` permet de spécifier le numéro du bit à écrire sachant que celui-ci commence à 0 pour le bit de poids le plus faible, c'est-à-dire le bit le plus à droite de la variable.
`bit` est égal à 0 ou 1 et permet donc de préciser la valeur du bit à écrire.
Le bit de `variable` précédemment contenu à l'emplacement ainsi écrit est évidemment écrasé par cette opération.

■ **bitSet() et bitClear()**

Ces deux fonctions font un peu double emploi avec la précédente. Elles permettent en effet d'écrire un bit à 1 (`bitSet`) ou un bit à 0 (`bitClear`) à n'importe quel emplacement dans une variable. `bitSet` est donc équivalent à `bitWrite (variable, n, 1)` tandis que `bitClear` est équivalent à `bitWrite (variable, n, 0)`.
Elles s'utilisent donc tout simplement de la façon suivante :

```
bitSet (variable, n) ;
bitClear (variable, n) ;
```

Comme précédemment, `n` permet de spécifier le numéro du bit à écrire sachant que celui-ci commence à 0 pour le bit de poids le plus faible, c'est-à-dire le bit le plus à droite de la variable.

2.8.3 Gestion des interruptions

Alors que de nombreux compilateurs font l'impasse sur la gestion des interruptions, celui de l'Arduino permet une utilisation relative de ces dernières qui, même si elle ne peut se comparer à ce qu'il est possible de faire lorsque l'on programme en langage machine, est tout de même mieux que rien.
Rappelons qu'une interruption est un évènement pouvant se produire n'importe quand pendant le déroulement d'un programme. Sous réserve que les interruptions soient gérées, il provoque alors l'abandon immédiat du programme en cours et l'exécution d'une portion de programme particulière, appelé programme de gestion de l'interruption. Lorsque l'exécution de ce programme d'interruption est terminée, le programme principal reprend son cours normalement, là où il avait été interrompu.
Les interruptions sont donc un moyen très performant pour surveiller un évènement extérieur comme le montre cet exemple simple. Soit une application quelconque pendant le déroulement de laquelle l'appui sur un poussoir doit provoquer un changement de comportement.
En programmation classique, on va devoir intercaler, tout au long du programme principal, des tests de ce poussoir sans être certain que l'on ne manquera pas un appui qui serait par exemple très bref.
En programmation sous interruption, on va câbler le poussoir de façon à ce qu'il agisse sur une entrée d'interruption. Dans ces conditions, il n'y aura plus besoin d'aucun test dans le programme principal et on ne risquera pas de manquer un appui sur le poussoir car, dans tous les cas, ce dernier provoquera une interruption qui provoquera nécessairement à son tour l'interruption du programme principal et son traitement par le programme d'interruption correspondant.
Tous les microcontrôleurs actuels, et ceux qui équipent l'Arduino ne font pas exception, disposent de multiples sources d'interruptions (plusieurs par ressources internes en fait) cependant, toutes ne sont pas gérées par le compilateur de l'Arduino ou, plus exactement, seules sont gérées les interruptions résultat d'un changement d'état d'une entrée numérique externe. Pour cela, quatre fonctions sont à votre disposition.

© Dunod – La photocopie non autorisée est un délit.

■ attachInterrupt()

Cette fonction permet de spécifier tout à la fois quelle va être l'entrée numérique à utiliser pour générer une interruption, quel va être le type de signal qui va provoquer l'interruption et enfin quelle va être la fonction appelée pour traiter cette interruption. Elle s'utilise de la façon suivante :

```
attachInterrupt (entrée, fonction, mode) ;
```

– `entrée` permet de définir l'entrée utilisée pour générer l'interruption au moyen d'un « nom » numérique qui est 0 pour l'entrée numérique 2 et 1 pour l'entrée numérique 3 sur les Arduino Uno. Sur l'Arduino Mega 2560, il existe en plus 2 pour l'entrée 21, 3 pour l'entrée 20, 4 pour l'entrée 19 et 5 pour l'entrée 18.
– `fonction` est le nom de la fonction qui sera exécutée lors d'une interruption. C'est donc elle qui contient le programme d'interruption évoqué ci-dessus. Cette fonction ne peut recevoir aucun paramètre et ne doit retourner aucun résultat.
– `mode` permet quant à lui de spécifier le type de signal qui provoquera l'interruption. Quatre valeurs sont admises et ont la signification suivante :
  - `LOW` provoquera une interruption lorsque l'entrée sélectionnée sera au niveau logique bas ;
  - `CHANGE` provoquera une interruption lorsque l'entrée sélectionnée changera d'état logique (de haut vers bas ou de bas vers haut peu importe) ;
  - `RISING` provoquera une interruption lorsque l'entrée sélectionnée passera du niveau logique bas au niveau logique haut ;
  - `FALLING` provoquera une interruption lorsque l'entrée sélectionnée passera du niveau logique haut au niveau logique bas.

> **Attention !**
> Dans la fonction appelée par une interruption `delay` ne fonctionne pas. Par ailleurs, `millis` cesse de s'incrémenter et les données éventuellement reçues par `Serial` (voir ci-après) sont perdues. Enfin, il faut déclarer comme `volatile` toute variable modifiée par la fonction de traitement de l'interruption.

■ detachInterrupt()

Cette fonction permet d'arrêter la génération d'une interruption à partir d'une entrée préalablement validée par `attachInterrupt`. Elle s'utilise donc tout simplement de la façon suivante :

```
detachInterrupt(entrée) ;
```

`entrée` correspond évidemment à l'entrée à partir de laquelle on ne souhaite plus que soit générée d'interruption. Les règles de définition de cette entrée sont les mêmes que pour `attachInterrupt`.

■ interrupts() et noInterrupts()

Tout microcontrôleur qui se respecte offre la possibilité d'autoriser ou d'interdire les interruptions. L'interdiction des interruptions peut en effet s'avérer utile lorsque

l'on exécute une portion de programme dont le timing doit être précisément respecté et qui, de ce fait, ne doit pas pouvoir être interrompu.
Dans le cas de l'Arduino, les interruptions sont autorisées par défaut car elles sont utilisées par certaines fonctions, telles que `delay`, `millis` ou bien encore `Serial` par exemple.
Si, pour une portion de programme très particulière, il est nécessaire de les interdire ; il suffit d'écrire juste avant celle-ci :

```
noInterrupt() ;
```

Pour valider à nouveau les interruptions une fois cette section de programme passée, il suffit évidemment d'écrire :

```
interrupts() ;
```

Compte tenu des fonctions qui sont interrompues lorsque l'on interdit les interruptions, cette possibilité doit cependant être utilisée avec précaution.

2.9 Gestion du port série

Les Arduino Uno disposent d'un port série asynchrone accessible via les entrées/sorties numériques 0 et 1. La ligne 0 correspond à la ligne Rx ou réception de données tandis que la ligne 1 correspond à Tx c'est-à-dire émission de données. Ce partage des fonctions signifie aussi que, si vous décidez d'utiliser le port série, les entrées/sorties 0 et 1 ne seront plus utilisables en tant qu'entrées/sorties numériques.
Par ailleurs, compte tenu du mode d'interfaçage de l'Arduino au PC via l'interface USB et la création d'un port COM virtuel sur ce dernier, il est également possible d'exploiter ce port depuis le PC. L'environnement de développement comporte d'ailleurs un moniteur de port série accessible via la rubrique **Moniteur série** du menu **Outils**.
L'Arduino Mega 2560 quant à lui dispose de trois ports série additionnels appelés Serial1, Serial2 et Serial3 (le port d'origine commun avec les Arduino Uno s'appelant toujours Serial sans numéro). Ils sont utilisables de la même façon que le seul port série des autres Arduino mais en remplaçant, dans les explications qui vont suivre, `Serial` par `Serialn` où `n` est le numéro du port concerné.
`Serial` ne s'utilise jamais seule mais fait appel à un certain nombre de fonctions que nous allons brièvement présenter ci-après, quitte à y revenir plus avant dans cet ouvrage lors de leur utilisation.

■ Serial.begin()

Cette fonction doit être appelée au moins une fois, généralement dans la fonction `setup()` du programme, afin de définir la vitesse utilisée sur la liaison série. La syntaxe est fort simple :

```
Serial.begin(vitesse) ;
```

© Dunod – La photocopie non autorisée est un délit.

`vitesse` peut prendre l'une des valeurs de vitesse normalisée suivante : 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 et 115200. Mais il est également possible de spécifier une vitesse non normalisée si nécessaire.

■ Serial.end()

Elle permet de désactiver l'interface série asynchrone de l'Arduino. Les entrées/sorties numériques 0 et 1 peuvent donc à nouveau être utilisées pour tout autre chose. La syntaxe est tout simplement :

```
Serial.end() ;
```

■ Serial.available()

Le port série des microcontrôleurs AVR qui équipent les différents Arduino dispose d'un tampon capable de mémoriser jusqu'à 128 caractères. Cette fonction permet de savoir combien de caractères ont été reçus et sont donc contenus dans ce tampon en attente de lecture.
Elle s'utilise sous la forme :

```
nombre = Serial.available() ;
```

`nombre`, qui est de type `int`, contient alors le nombre de caractères reçus, et non encore lus par le programme, contenus dans le tampon du port série.

■ Serial.read ()

Comme son nom le laisse supposer cette fonction lit le premier caractère disponible dans le tampon de réception et donc le premier caractère à avoir été reçu. Elle s'utilise sous la forme :

```
caractère = Serial.read() ;
```

`caractère`, qui est nécessairement de type `int`, contient alors le premier caractère disponible dans le tampon de réception ou -1 si aucun caractère n'est disponible.

> **Attention !**
> Le tampon de réception est géré automatiquement, c'est-à-dire que chaque caractère lu est automatiquement enlevé de ce tampon et que le résultat de `Serial.available` évolue donc en conséquence.

■ Serial.flush()

Sur les versions de l'environnement de développement précédant la version 1.0, cette fonction permettait tout simplement de vider le tampon de réception afin qu'il soit prêt à recevoir de nouvelles données.
Depuis la version 1.0 de l'environnement de développement, cette fonction a pour seul effet d'attendre la fin de l'émission de données *via* le port série.
Dans les deux cas, elle s'utilise sous la forme suivante :

```
Serial.flush() ;
```

■ Serial.print()

Cette fonction est certainement l'une des plus riches, en termes de paramétrage, qui soit disponible avec `Serial`. Elle permet de faire sortir, sur la liaison série, une ou plusieurs données et de préciser leur mode de codage afin de faciliter leur affichage sur un terminal, sur un afficheur alphanumérique à cristaux liquides ou bien encore sur tout dispositif acceptant en entrée du code ASCII. Elle assure en effet automatiquement la conversion des données qu'elle traite en leur code ASCII pour permettre leur affichage de façon lisible.
Ainsi par exemple :

- `Serial.print (25)` fera afficher 25 ;
- `Serial.print (1.23456)` fera afficher 1.23 (car par défaut elle affiche les nombres en virgule flottante avec deux décimales, mais cela peut être modifié) ;
- `Serial.print (« Arduino »)` fera afficher Arduino ;
- et ainsi de suite.

Cette fonction s'utilise en fait sous l'une des deux formes suivantes :

```
Serial.print (donnée) ;
Serial.print (donnée, format) ;
```

Dans le premier cas la donnée, si elle est numérique, est considérée comme décimale. Si c'est une chaîne de caractères, elle est affichée telle quelle et si c'est un nombre en virgule flottante, elle est affichée avec deux décimales.

Dans le second cas, l'information `format` permet de préciser la base de numération pour les données numériques. Elle peut alors prendre une des valeurs suivantes : `BIN` pour du binaire, `OCT` pour de l'octal (de moins en moins utilisé), `HEX` pour de l'hexadécimal, `DEC` pour du décimal ou `BYTE` pour afficher le caractère dont le code ASCII correspond à la donnée. Si la donnée est exprimée en virgule flottante, `format` ne peut qu'être une valeur numérique comprise entre 0 et 7 qui spécifie alors le nombre de décimales à afficher.

Voici quelques exemples qui montrent clairement le rôle du paramètre format :

- `Serial.print (75)` donne 75 ;
- `Serial.print (75, DEC)` donne aussi 75 puisque la notation décimale est utilisée par défaut par `Serial.print` ;
- `Serial.print (75, BIN)` donne 100101 qui est la représentation en binaire de 75 ;
- `Serial.print (75, HEX)` donne 4B qui est la représentation en hexadécimal de 75 ;
- `Serial.print (75, BYTE)` donne K (la lettre K) parce que le code ASCII de cette lettre est égal à 75 ;
- `Serial.print (1.23456, 0)` donne 1 puisque l'on a spécifié aucune décimale ;
- `Serial.print (1.23456)` donne 1.23456 puisque l'on a spécifié 5 décimales.

© Dunod – La photocopie non autorisée est un délit.

> **Attention !**
> Cette fonction permet de toucher du doigt une notion qui perturbe parfois les débutants en programmation qui est de ne pas confondre la valeur d'une donnée (75 dans nos exemples) et sa représentation en ASCII, c'est-à-dire sous une forme lisible par l'homme. Dans tous les exemples ci-dessus (sauf ceux avec les nombres décimaux), notre donnée vaut toujours 75 en décimal dans le cadre du programme qui la manipule, et ce n'est que sa représentation qui change en fonction du paramètre `format` de la fonction `Serial.print`.

■ Serial.println()

Cette fonction s'utilise exactement comme `Serial.print` et dispose des mêmes options de formatage mais elle termine chaque ligne de données affichées par les caractères saut ligne et retour chariot.

En d'autres termes, si vous utilisez plusieurs `Serial.print` les uns après les autres, les caractères se suivront sur la même ligne tandis qu'avec plusieurs `Serial.println`, ils se suivront sur des lignes distinctes.

■ Serial.write()

Tout comme la fonction `Serial.print`, cette fonction a pour effet de faire sortir une ou plusieurs données sur la liaison série mais, contrairement à la fonction `Serial.print`, elle ne réalise aucune conversion de format. Les données sortent donc sous leur forme binaire brute, comme une suite d'octets qui ne sont en aucun cas convertis.

Elle peut s'utiliser sous l'une des trois formes suivantes :

```
Serial.write (donnée) ;
Serial.write (chaîne) ;
Serial.write (tableau, longueur) ;
```

Dans le premier cas, la fonction fait sortit l'unique donnée spécifiée. Dans le second cas, elle fait sortir les caractères qui composent la chaîne, l'un après l'autre, chaque caractère étant représenté par son code ASCII. Dans le troisième cas enfin, elle fait sortir l'une après l'autre les données contenues dans le tableau (ou donnée indexée si vous préférez) dont la taille est spécifiée par le paramètre `longueur`.

Quelques autres fonctions de gestion de cette interface série sont disponibles mais, à ce stade de l'ouvrage, il n'est pas nécessaire de vous les présenter car il est peu probable que vous ayez à y faire appel lors de vos premières manipulations avec l'Arduino.

3 • LES CARTES D'INTERFACE OU SHIELDS DE BASE

Comme nous l'avons expliqué au début de cet ouvrage, un des intérêts majeurs de l'Arduino est qu'il existe aujourd'hui sur le marché une multitude de cartes d'interfaces, appelées shields ou boucliers ou bien encore blindages dans de très mauvaises traductions françaises que nous n'utiliserons pas, capables de couvrir la majorité des besoins d'une application. Ces cartes ont une taille généralement voisine de celle de l'Arduino et s'enfichent directement sur les connecteurs périphériques de cette dernière.

Pour peu que leurs fabricants aient pris quelques précautions, ou que l'on ait fait le bon choix en les assemblant, pour celles qui sont vendues en kit, il est même possible d'empiler plusieurs cartes les unes sur les autres si une seule ne suffit pas.

Dans ce chapitre, nous vous proposons de découvrir les shields de base, c'est-à-dire ceux supportant des composants ou fonctions simples mais néanmoins utiles, voire même indispensables. Ces shields pourront être utilisés tels quels dans certaines applications, mais vous pourrez aussi y faire appel pour mettre au point vos propres circuits ou bien encore pour vous initier de manière pratique à la programmation de l'Arduino.

Cette initiation sera d'autant plus facile qu'à partir du chapitre 4 nous expliquerons comment interfacer l'Arduino avec la majorité des éléments que l'on est susceptible d'utiliser dans une application : poussoirs, commutateurs, LED, afficheurs, moteurs, relais, etc.

3.1 Les shields prototypes

Ce sont évidemment les premiers shields qui ont été développés car ce sont tout à la fois les plus simples à concevoir et les premiers que l'on utilise lorsque l'on cherche à réaliser une application.

Théoriquement, un shield prototype ne supporte aucun composant mais comporte seulement une zone pastillée destinée à recevoir les éléments de l'interface que l'on souhaite développer. En fait, pour rester en adéquation avec le concept Arduino, ces shields prototypes ne peuvent pas être conçus n'importe comment, même si l'expérience montre que certains sont mieux pensés que d'autres.

© Dunod – La photocopie non autorisée est un délit.

Nous vous proposons donc de découvrir ci-dessous les éléments essentiels à prendre en compte lors du choix d'un tel shield, ce qui nous conduira à vous présenter deux exemples de shields prototypes actuellement commercialisés : un modèle très bien conçu que nous retiendrons pour la suite de nos expérimentations et un modèle à éviter ; preuve, s'il en était besoin, qu'un tel shield n'est pas aussi trivial qu'on pourrait le penser.

3.1.1 Le minimum vital

Un shield prototype doit évidemment disposer d'une zone pastillée au pas de 2,54 mm destinée à l'implantation des composants de l'interface à tester. Pour que le câblage y soit le plus aisé possible, il est souhaitable qu'au moins une bande de pastilles soit reliée au + 5 volts de l'Arduino et une autre à la masse. C'est fort heureusement le cas de tous les modèles que nous avons eus entre les mains.

De part et d'autre de ce shield se trouvent les connecteurs véhiculant les signaux d'interface ; connecteurs destinés à s'enficher dans leurs homologues de la carte Arduino de base.

Afin que l'on puisse empiler plusieurs shields les uns sur les autres, il est souhaitable que ces connecteurs soient de type mâle-femelle, comme indiqué figure 3.1, et non de type mâle simple, comme indiqué figure 3.2. Hélas, un certain nombre de fournisseurs de cartes prototypes ne l'ont semble-t-il pas compris et continuent de proposer leurs cartes avec les connecteurs de la figure 3.2.

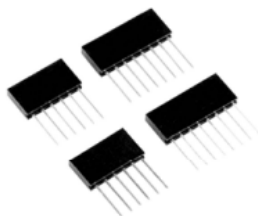

Figure 3.1 – Les connecteurs à utiliser sur une carte prototype.

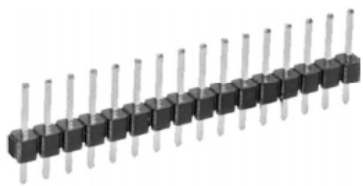

Figure 3.2 – Les connecteurs à éviter sur une carte prototype.

Avec les premières versions d'Arduino, la mise en place d'une carte prototype masquait un élément très important qui est le poussoir de reset. Même si l'on peut

éventuellement s'en passer, puisque le fait d'arrêter puis de remettre sous tension l'Arduino réalise un reset automatique ; le fait de disposer d'un poussoir de reset en phase de développement d'un programme est tout de même quelque chose de très confortable. Il est rare en effet qu'un programme et son interface associée fonctionnent du premier coup, et le recours au reset est donc souvent nécessaire. Ce problème est aujourd'hui résolu avec l'Arduino Révision 3 dont le poussoir de reset a été ramené en bordure de carte. Il est donc accessible même si des shields sont enfichés sur l'Arduino. Comme de nombreuses cartes ont été conçues avant la naissance de l'Arduino Révision 3, elles disposent souvent encore d'un poussoir de reset qui duplique tout simplement celui de la carte de base.

Zone pastillée avec bandes d'alimentation et connecteurs mâle-femelle constituent donc le minimum vital d'une carte prototype bien conçue. Cependant, toutes les cartes du commerce proposent des éléments supplémentaires. Voyons ce qu'il en est.

3.1.2 Les éléments optionnels

Même si nous n'en avons pas parlé jusqu'à présent, toutes les cartes Arduino de base supportent un connecteur, appelé ICSP, qui permet si nécessaire une programmation directe du microcontrôleur qui les équipe au moyen d'un programmateur destiné aux circuits de la famille AVR d'Atmel.

Ce connecteur étant partiellement masqué par la mise en place de la carte prototype ; certaines d'entre elles comportent un emplacement qui lui est destiné. C'est à notre avis une perte de place sur une telle carte car l'usage d'un tel connecteur reste marginal, surtout au stade de développement d'une application où l'on programme l'Arduino à partir de son outil de développement et donc via le port USB de liaison au PC. La présence d'un emplacement pour un tel connecteur n'est donc pas à notre avis un critère de choix d'une carte prototype. S'il est présent tant mieux ; s'il est absent, cela ne porte pas à conséquence.

Toutes les cartes prototypes disposent généralement d'un emplacement permettant de câbler une ou deux LED associées à leurs résistances séries de limitation de courant. Cela n'est évidemment pas indispensable mais peut rendre service dans certaines applications où l'on souhaite connaître l'état de signaux logiques. Au pire l'une d'entre elles peut servir de simple témoin de mise sous tension.

Outre le poussoir de reset évoqué ci-dessus, la majorité des cartes prototypes dispose également d'un autre poussoir et de sa résistance de rappel au +5 volts. Ce poussoir peut être utilisé comme organe d'entrée de signal logique pour votre application si nécessaire.

3.1.3 Bonnes et mauvaises cartes prototypes

Au vu de l'exposé qui précède, vous devez déjà avoir une idée assez précise de la façon dont doit être conçue une carte prototype. Nous vous proposons donc d'en choisir une que vous pourrez utiliser pour câbler quelques-uns des circuits présentés dans la suite de cet ouvrage pour vous initier à la programmation de l'Arduino.

© Dunod – La photocopie non autorisée est un délit.

En ce qui nous concerne, nous avons retenu le modèle ProtoShield de Sparkfun (*www.sparkfun.com*) distribué en France par Lextronic (*www.lextronic.fr*). Outre son prix particulièrement attractif, cette carte satisfait tous nos critères de base et dispose, en éléments optionnels, de deux LED et du poussoir supplémentaire que nous venons d'évoquer.

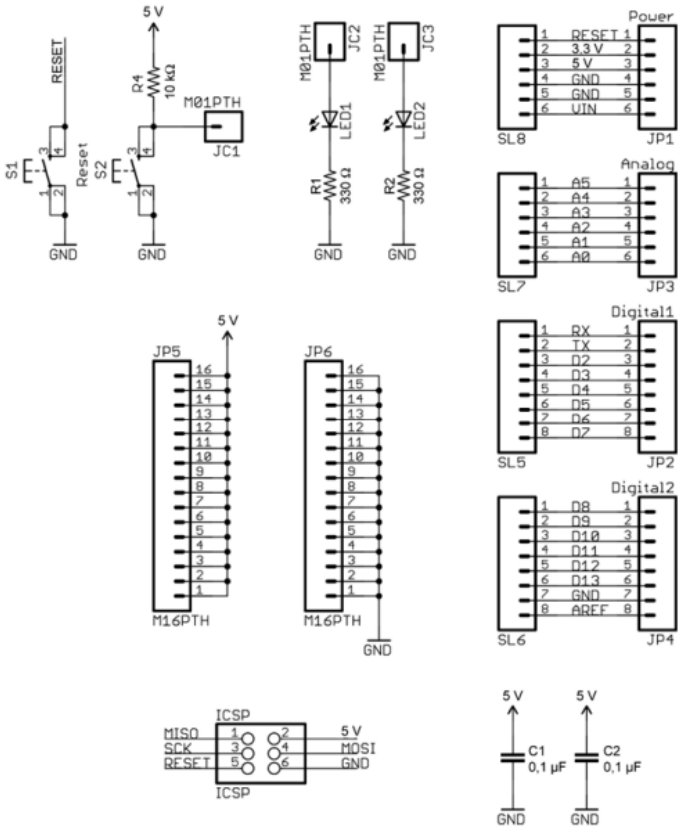

Figure 3.3 – Schéma de la carte ProtoShield de Sparkfun (doc. Sparkfun).

Elle est fournie en kit mais son montage est évidemment d'une grande simplicité. Son schéma vous est présenté figure 3.3 tandis que la figure 3.4 montre l'allure de la carte prête à l'emploi.

Si vous envisagez d'utiliser cette carte pour développer vos propres interfaces, nous n'avons rien à ajouter. Si par contre vous souhaitez l'utiliser pour vous initier à la

programmation de l'Arduino, au moyen des exemples fournis dans la suite de cet ouvrage par exemple, nous vous conseillons de l'équiper d'une plaque de câblage rapide (ou de prototypage si vous préférez).

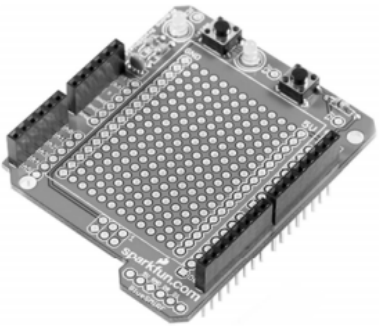

Figure 3.4 – La carte ProtoShield de Sparkfun (doc. Sparkfun).

En effet, même si les trous métallisés de la zone pastillée de la carte sont de bonne qualité, ils supportent assez mal les opérations de soudage et de dessoudage successives que l'on doit faire lorsque l'on essaye de multiples circuits. Une plaque de câblage rapide perforée est donc mieux adaptée pour cela.

Mais, compte tenu du faible espace disponible entre les deux rangées de connecteurs de la carte, il faut faire appel à des plaques de câblage rapides spécialement prévues à cet effet, disponibles également chez Sparkfun (*www.sparkfun.com*) ou en France chez Lextronic (*www.lextronic.fr*). Ces plaques peuvent être collées sur la carte prototype grâce à l'adhésif double face dont elles sont munies, comme indiqué figure 3.5 sur laquelle on voit une carte prototype ainsi équipée enfichée sur une carte Arduino Uno.

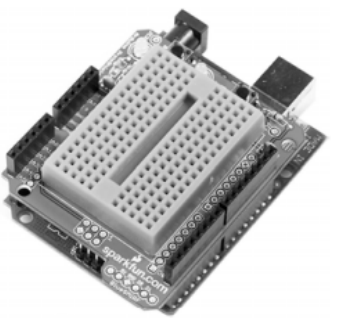

Figure 3.5 – Une carte prototype munie de sa plaque de câblage rapide (doc. Sparkfun).

© Dunod – La photocopie non autorisée est un délit.

Si votre budget le permet, approvisionnez en même temps quelques straps flexibles mâles-mâles spécialement prévus pour ces plaquettes ou réalisez-les en coupant diverses longueurs de fil de câblage rigide de 6/10 de millimètre (mm) de diamètre que vous dénuderez sur 10 mm environ à chaque extrémité. C'est moins beau que les straps du commerce mais ça fonctionne aussi bien !

Compte tenu du nombre de cartes prototypes disponibles sur le marché, vous pouvez choisir un autre modèle que la carte Sparkfun préconisée mais nous vous conseillons d'éviter celle vendue sous la dénomination Proto Shield V4.0, dont nous n'avons pas pu identifier le fabricant mais que l'on trouve notamment chez Radiospares (*www.radiospares.fr*) et donc l'aspect est visible figure 3.6.

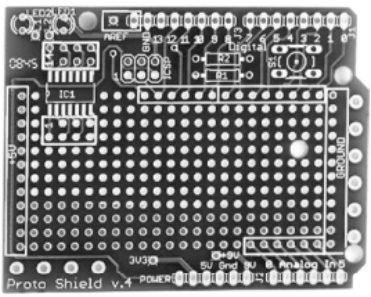

Figure 3.6 – Une carte prototype mal conçue (doc. Radiospares).

En effet, outre le fait que cette carte soit livrée avec des connecteurs mâles simples ce qui, nous l'avons vu ci-dessus, interdit son empilement ; elle comporte une aberration au niveau du tracé de son circuit imprimé qui fait qu'une piste nue, véhiculant le + 5 volts, vient toucher le corps métallique de la prise USB de l'Arduino qui est bien entendue reliée… à la masse. Le court-circuit est assuré, sauf à mettre du ruban adhésif sur ce connecteur ce qui, vous en conviendrez, n'est pas très professionnel !

3.1.4 Un shield qui n'en est pas un

Avant de passer à l'étude d'autres shields expérimentaux, il nous semble important d'attirer votre attention sur une solution susceptible de vous intéresser, toujours dans le but de vous initier à l'Arduino ou de mettre au point vos propres cartes d'interface. Comme vous pouvez le constater à l'examen de la figure 3.7, il ne s'agit pas d'un shield mais tout simplement d'une plaque en plastique moulée comportant deux cavités. L'une d'elles peut recevoir, et maintenir en place au moyen de deux clips, une carte Arduino Uno, tandis que l'autre peut recevoir une plaque de câblage rapide de dimensions raisonnables, et en tout cas beaucoup plus grande que celle pouvant prendre place sur un shield classique. Ce produit, vendu à un prix dérisoire, est au catalogue de Sparkfun (*www.sparkfun.com*) décidément très actif dans le domaine des produits destinés à l'Arduino, sous la dénomination Arduino and Breadboard Holder.

Figure 3.7 – L'Arduino and Breadboard Holder de Sparkfun (doc. Sparkfun).

L'intérêt de cette approche est de permettre le câblage expérimental d'interfaces plus complexes que celles permises par les shields prototypes classiques en raison de la place disponible sur la plaque de câblage rapide.
En contrepartie, comme aucune connexion directe n'existe entre les entrées/sorties de l'Arduino et la plaque de câblage, ce dernier est nécessairement un peu plus pénible, surtout si de nombreux signaux sont utilisés.

3.2 Un shield idéal pour la formation

Si vous désirez vous former à l'Arduino, ou former à votre tour d'autres utilisateurs potentiels, sans devoir recourir sans cesse au fer à souder ou aux straps de câblage rapide, il existe une solution plus pratique que celle consistant à faire appel à un shield prototype tels ceux que nous venons de présenter. Elle passe tout simplement par l'utilisation du Danger Shield qui, contrairement à ce que son nom laisse supposer, ne présente aucun danger d'utilisation !
Ce shield, commercialisé lui aussi par Sparkfun (*www.sparkfun.com*), est disponible en France chez Lextronic (*www.lextronic.com*). Il supporte un certain nombre d'éléments, disparates au premier abord, mais indispensables pour qui veut apprendre à bien utiliser un Arduino. On y trouve en effet :

- un afficheur à LED 7 segments relié à un registre à décalage de type 74HC595 ;
- deux LED libres et une LED témoin d'alimentation ;
- trois poussoirs ;
- trois potentiomètres rectilignes ;
- une photorésistance LDR (*Light Dependent Resistor* ou résistance sensible à la lumière) ;
- un capteur de température intégré de type TMP36 ;
- un buzzer piézoélectrique utilisable en tant que tel ;
- un autre buzzer piézoélectrique utilisable en tant que détecteur de choc ;
- un poussoir de reset moins utile avec l'avènement de l'Arduino Révision 3.

© Dunod – La photocopie non autorisée est un délit.

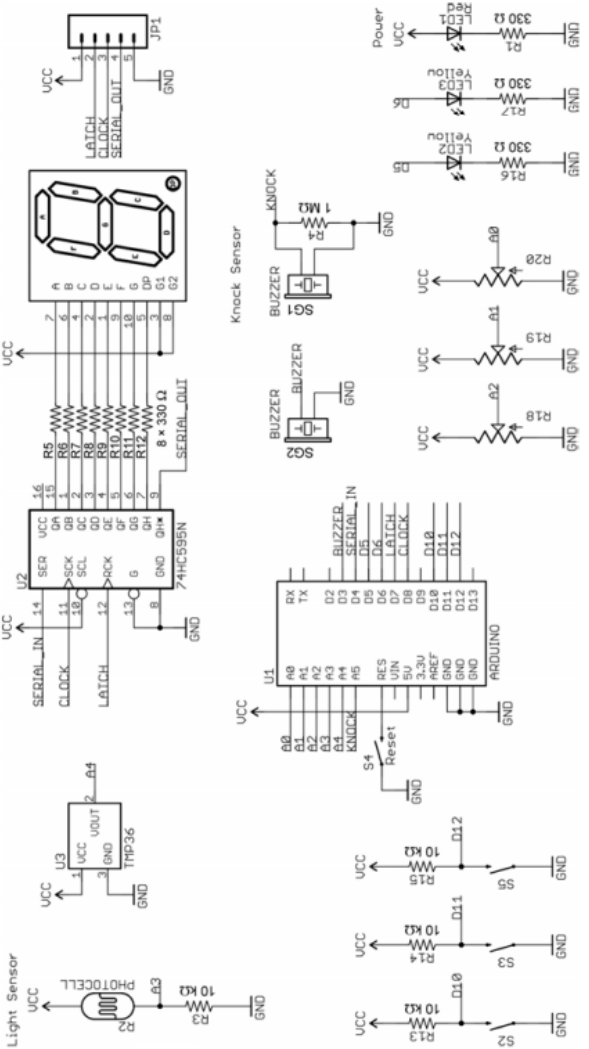

Figure 3.8 – Schéma de la carte Danger Shield de Sparkfun (doc. Sparkfun).

Il est donc très peu probable que cette carte soit utilisée telle qu'elle dans une de vos applications mais, par contre, nous verrons à partir du chapitre suivant qu'elle permet de s'initier très facilement à l'interfaçage de l'Arduino sans avoir à réaliser aucun câblage puisqu'elle supporte nombre d'éléments que l'on rencontre habituellement lors du développement d'un programme.

Son schéma vous est présenté figure 3.8 et s'avère fort simple. Les LED sont équipées de leurs résistances séries de limitation de courant tandis que les poussoirs sont associés à leurs résistances de tirage au niveau haut (voir chapitre suivant si nécessaire à ce sujet).

Le capteur de température TMP36 est prêt à l'emploi et ne nécessite aucun composant externe pour fonctionner ; la carte se charge donc seulement de l'alimenter. L'afficheur 7 segments quant à lui est relié à la sortie du registre à décalage de type 74HC595 afin de permettre sa commande par l'Arduino avec seulement trois fils comme nous le verrons au chapitre 6.

Compte tenu de l'encombrement des composants qu'il supporte, ce shield est plus grand que la carte Arduino de base comme le montre la figure 3.9. Néanmoins, il s'enfiche sur cette dernière sans problème afin d'éliminer tout recours à du câblage externe.

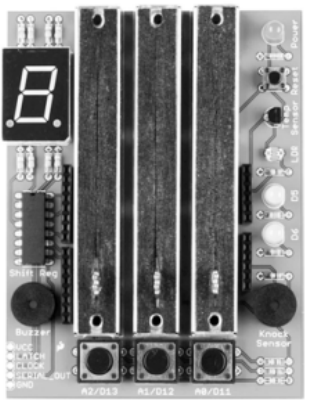

Figure 3.9 – La carte Danger Shield de Sparkfun (doc. Sparkfun).

3.3 Un shield vide mais fort utile

Le shield que nous allons présenter maintenant peut être utilisé de deux façons différentes à savoir dans une application réelle ou dans un but de formation comme le Danger Shield que nous venons de présenter.

© Dunod – La photocopie non autorisée est un délit.

Ce shield, baptisé Arduino Sensor Shield, ne supporte en effet aucun composant mis à part une LED, trois résistances et le poussoir de reset désormais bien connu. Par contre, comme vous pouvez le constater figure 3.10, il est littéralement recouvert de connecteurs de tous poils.

En effet, même si les cartes de base Arduino disposent sur leur pourtour de connecteurs d'interface, ceux-ci sont très vite saturés, soit parce que l'on a enfiché un ou plusieurs shields sur la carte de base, soit parce que l'on doit faire appel à des fonctions diverses partagées par les mêmes connecteurs (bus I2C et ports parallèles par exemple ou bien encore liaison série asynchrone et ports parallèles).

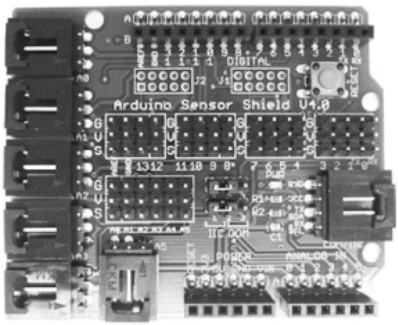

Figure 3.10 – La carte Arduino Sensor Shield (doc. Emartee).

En outre, de nombreux éléments externes reliés à l'Arduino, capteurs de température, de lumières, potentiomètres, etc. nécessitent bien souvent une alimentation.

La carte Arduino Sensor Shield propose donc plusieurs types de connecteurs à trois contacts, chacun d'entre eux comprenant une ligne d'entrée/sortie logique ou analogique de l'Arduino, la masse et l'alimentation + 5 volts. Comme vous pouvez le constater à l'examen de la figure 3.10, ces connecteurs sont soit de simples picots mâles au pas de 2,54 mm soit des connecteurs verrouillables, eux aussi au pas de 2,54 mm.

Sur la version 4.0 de la carte un connecteur baptisé port de communication peut véhiculer soit un bus I2C soit une liaison série asynchrone selon la position de straps placés derrière lui. Sur la version 5.0 de cette même carte, ces straps ont disparu et deux ports de communication sont présents : un pour le bus I2C et l'autre pour une liaison série asynchrone.

Cette carte peut donc être utilisée dans une application finalisée afin d'en faciliter le câblage, mais elle peut aussi être utilisée à des fins de formation. En effet, son promoteur, la société Seeed Studio Works (*www.seeedstudio.com*) commercialise aussi ce qu'elle appelle des « bricks » qui sont de minuscules cartes, munies chacune d'un élément bien précis tel que, par exemple, potentiomètre, LED, poussoir, capteur de température, cellule photo électrique, etc. Ces bricks sont tous munis

d'un connecteur à trois points qui peut, vous l'aurez compris, être relié immédiatement aux connecteurs correspondants de la carte Sensor Shield.
La figure 3.11 montre ainsi quelques-uns de ces bricks afin que vous ayez une idée plus précise de ce à quoi vous pouvez vous attendre.

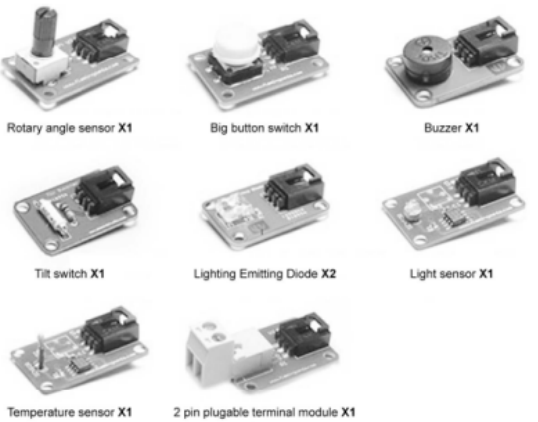

Figure 3.11 – Quelques « bricks » utilisables avec l'Arduino Sensor Shield.

Au moment où ces lignes sont écrites, près d'une vingtaine de « bricks » sont disponibles et couvrent quasiment tous les besoins en matière de formation puisqu'ils proposent des éléments plus nombreux que ceux regroupés sur la carte Danger Shield. Divers kits sont également proposés regroupant la carte Arduino Sensor Shield et des assortiments plus ou moins variés de « bricks ».
Ces ensembles sont donc très intéressants pour de la formation ou du prototypage rapide d'autant que le prix des kits reste inférieur à celui des seuls composants qui seraient nécessaires pour les réaliser. Au moment où ces lignes sont écrites, ces éléments ne sont pas encore distribués en France mais vous les trouverez par exemple chez Seeed Studio Works (*www.seeedstudio.com*) ou bien encore chez Emartee (*www.emartee.com*) qui pratiquent tous deux la vente en ligne.

3.4 Un shield pour y voir clair

De nos jours, quasiment toutes les applications à base de microcontrôleur qui dialoguent avec l'utilisateur autrement que par de simples chiffres utilisent des afficheurs alphanumériques qui sont généralement des modèles comportant un minimum de 2 lignes de 16 ou 20 caractères.
Lorsqu'ils sont présents, ces afficheurs proposent bien souvent des systèmes de menus pour paramétrer telle ou telle fonction de l'application, menus qui sont

© Dunod – La photocopie non autorisée est un délit.

généralement gérés au moyen des touches désormais classiques que sont : flèche à gauche, à droite, en haut et en bas et touche de validation.
Même si la réalisation d'une interface munie d'un afficheur de ce type et des touches qui vont avec ne présente aucune difficulté, elle nécessite de recourir au fer à souder et au circuit imprimé spécifique, sauf à faire appel au LCD – Keypad Module qui est justement un shield prévu à cet effet.

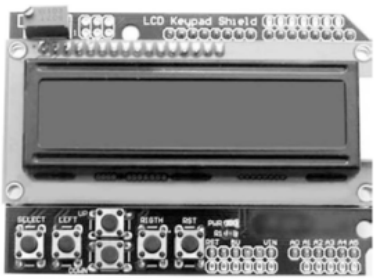

Figure 3.12 – Le shield LCD – Keypad Module (doc. Electronic Kits Zone).

Comme vous pouvez le constater figure 3.12, ce shield supporte un afficheur à cristaux liquides alphanumérique standard de 2 lignes de 16 caractères, rétro éclairé de surcroît. Il est associé à quatre touches disposées en croix (les touches directionnelles évoquées ci-dessus) et à une touche qui pourra faire office de touche de validation.
Le traditionnel poussoir de reset, nettement déporté des autres touches pour éviter toute confusion fâcheuse, est également présent.
Comme on peut le constater à l'examen du schéma de ce module, visible figure 3.13, l'afficheur LCD est utilisé en mode 4 bits (voir chapitre 4) ce qui ne pose aucun problème de gestion par l'Arduino car la bibliothèque prévue à cet effet supporte ce mode de connexion.
Les touches par contre font appel à un mode de connexion original puisqu'elles sont toutes reliées ensembles, via des diviseurs de tension à résistances, à une seule entrée analogique. Nous verrons au chapitre 4 que cette façon de faire permet de minimiser le nombre d'entrées/sorties nécessaires pour lire un clavier, ce qui s'avère fort utile dans le cas des Arduino Uno dont le nombre de lignes de ce type est assez limité.
Compte tenu du fait que ce shield est utilisé pour dialoguer avec l'utilisateur de l'application qu'il équipe, il doit évidemment se trouver « sur le haut de la pile » des shields qui la composent. Il n'est donc pas muni des traditionnels connecteurs mâles-femelles car cela n'aurait aucun sens.
Au moment où ces lignes sont écrites, ce shield n'est pas ou est mal distribué en France. Par contre, il est très facile de le trouver chez les différents revendeurs de shields pour Arduino présents sur Internet, et ce d'autant qu'il semble être fabriqué par plusieurs sources d'origine « extrême orientale »…

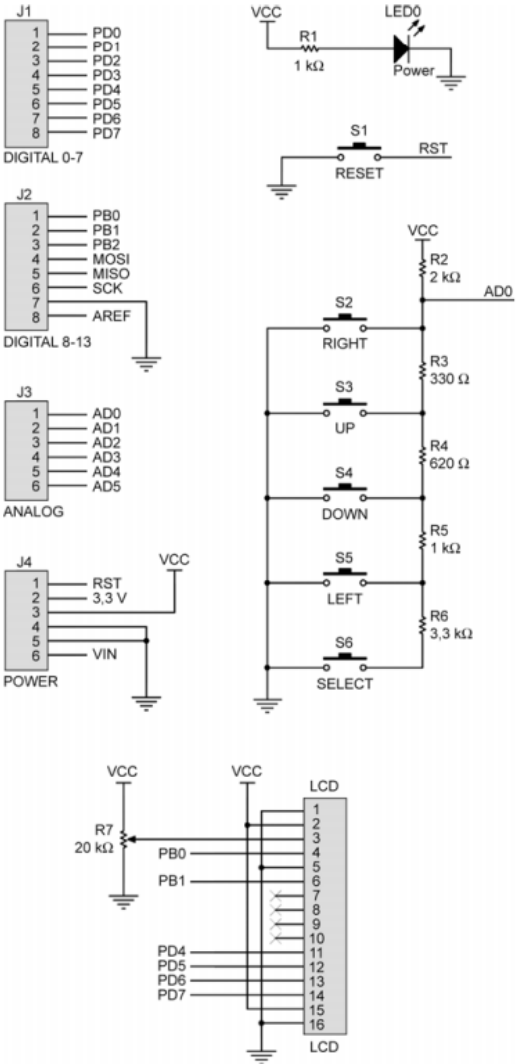

Figure 3.13 – Schéma du shield LCD – Keypad Module (doc. Electronic Kits Zone).

Celui que nous avons utilisé dans le cadre de cet ouvrage et qui a servi de base aux figures 3.12 et 3.13 provient de chez Electronic Kits Zone (*www.ekitszone.com*).

© Dunod – La photocopie non autorisée est un délit.

4 • ENTRÉE/SORTIES NUMÉRIQUES PARALLÈLES : POUSSOIRS, CLAVIERS, LED, RELAIS ET AFFICHEURS

Quel que soit le microcontrôleur que l'on utilise dans une application, et en ce domaine l'Arduino ne se singularise pas, il passe le plus clair de son temps à communiquer avec le monde extérieur par le biais de ce que l'on désigne sous le nom générique d'entrées/sorties. Bien sûr, ces dernières varient fortement d'une application à une autre. Ainsi, si votre microcontrôleur est l'automate de gestion d'un lave-vaisselle, ses entrées seront des contacts de capteurs divers, et ses sorties des commandes de moteurs et d'électrovannes. Par contre, s'il s'agit d'un thermostat programmable, il utilisera un clavier et un afficheur destinés à communiquer avec l'utilisateur, ainsi qu'une sonde de température et un relais de commande de la chaudière.

Cette diversité des entrées/sorties n'a cependant rien d'inquiétant ni de difficile à maîtriser, en effet, la majorité d'entre elles fait appel à des schémas de base classiques et à leurs programmes associés que nous avons regroupés dans la suite de ce chapitre.

Les solutions que nous allons vous présenter n'ont cependant pas la prétention d'être les seules utilisables ou d'être les meilleures. En effet, ce qui fait la force d'un microcontrôleur c'est qu'il existe bien souvent une multitude de moyens différents pour parvenir au même résultat. C'est à vous de déterminer celui qui vous convient le mieux en utilisant les exemples présentés ici comme tremplins.

Ce chapitre, consacré au dialogue avec le monde extérieur, traite des entrées/sorties parallèles. Les entrées parallèles sont utilisées pour connaître l'état de divers éléments : commutateurs, poussoirs, etc., pour vérifier la présence ou l'absence de tension sur certains dispositifs au moyen, par exemple, de photocoupleurs d'isolement, ou bien encore pour scruter un clavier de commande mis à disposition de l'utilisateur de l'application.

Les sorties parallèles quant à elles sont utilisées pour commander des LED, des relais, des afficheurs, qu'ils soient à base de diodes électroluminescentes ou bien encore de cristaux liquides.

Dans les deux cas, ce sont les interfaces externes les plus faciles à gérer ; il est donc logique que vous commenciez avec elles vos essais. Précisons à ce propos que la majorité des expérimentations proposées dans ce chapitre peut être réalisée quasi-

© Dunod – La photocopie non autorisée est un délit.

ment sans câblage en utilisant soit le Danger Shield, soit l'Arduino Sensor Shield associé à un certain nombre de « bricks » comme nous l'avons vu au chapitre 3. Si un peu de câblage ne vous fait pas peur, l'utilisation d'un shield prototype ou de l'Arduino Breadboard Holder, également présentés au chapitre 3, est également envisageable.

4.1 Les entrées numériques parallèles

Les entrées parallèles sont parmi les plus utilisées sur un microcontrôleur. En effet, que ce soit pour lire l'état de poussoirs, d'interrupteurs ou de claviers ou bien encore pour vérifier la présence de tension en tel ou tel point d'une application, ce sont les plus simples à mettre en œuvre, d'autant qu'elles ne requièrent souvent que très peu de composants externes. Il est donc logique de leur laisser la première place dans cet exposé.

4.1.1 Poussoirs, commutateurs, capteurs de type tout ou rien

La lecture de l'état de poussoirs, de commutateurs, de roues codeuses (qui ne sont que des commutateurs un peu particuliers) ou de capteurs de type tout ou rien, c'est-à-dire de capteurs qui se contentent de fermer ou d'ouvrir un contact (ILS, ampoule au mercure, etc.) est d'une extrême simplicité, que ce soit avec un Arduino ou avec n'importe quel autre type de microcontrôleur d'ailleurs.

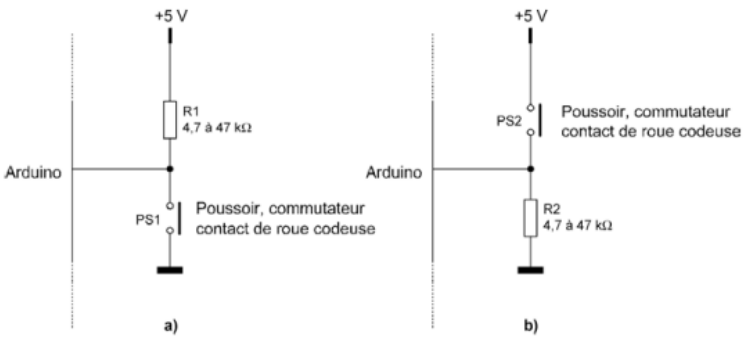

Figure 4.1 – Interface avec un poussoir, un commutateur ou plus généralement tout contact.

Il suffit en effet de relier ce dernier entre une entrée parallèle et la masse comme schématisé figure 4.1a ou entre une entrée parallèle et le +5 volts comme indiqué figure 4.1b. Dans le cas de la figure 4.1a, l'entrée est par ailleurs ramenée au niveau logique haut par une résistance de tirage ou « pull up » pouvant aller jusqu'à 47 ou 100 kΩ si vous avez des impératifs de faible consommation.

Si tel n'est pas le cas, et si vous travaillez dans un milieu susceptible d'être perturbé (voisinage de moteurs par exemple), il est préférable d'utiliser une résistance de valeur plus faible (typiquement 4,7 ou 10 kΩ). En effet, il est plus facile de parasiter un niveau logique haut maintenu par une résistance de 100 kΩ qu'un niveau du même type maintenu par une résistance de 10 kΩ ou de 4,7 kΩ.

Tant que le commutateur est ouvert, l'entrée est vue comme étant au niveau logique haut. Lorsqu'il est fermé, l'entrée est vue comme étant au niveau logique bas. Il suffit donc de lire l'état de la ou des lignes du port parallèle concerné pour connaître celui de l'interrupteur ou du commutateur, ce qui se fait très facilement comme nous allons le voir dans un instant.

Notez que, depuis l'apparition de la définition `INPUT_PULLUP` au niveau de la fonction pinMode(), ce qui est possible à partir de l'environnement de développement en version 1.0.1, le schéma de la figure 4.1a n'est plus indispensable. En effet, si vous définissez les entrées avec le type `INPUT_PULLUP`, la résistance de tirage au niveau haut contenue dans le microcontrôleur ATmega qui équipe l'Arduino est activée et joue donc le rôle de la résistance R1 de la figure 4.1a. La résistance R1 devient donc inutile dans un tel cas.

Dans le cas de la figure 4.1b tout ce que nous venons d'écrire reste valable mais c'est évidemment le contraire qui se produit, à savoir un niveau logique bas lorsque l'interrupteur est ouvert et un niveau haut lorsqu'il est fermé.

> **Attention !**
> Dans le mode de câblage de la figure 4.1b, il ne faut pas utiliser la définition `INPUT_PULLUP` au niveau de la fonction pinMode() car la résistance de tirage interne du microcontrôleur formerait alors un pont diviseur de tension parasite avec R2, ce qui conduirait à la génération de niveaux logiques pouvant être invalides et, donc, mal interprétés.

Ceci étant, et si vous avez le choix, il est toujours préférable de choisir le schéma a plutôt que le b. En effet, si l'interrupteur ou le capteur doit être déporté hors de la carte d'interface de l'Arduino, il vaut mieux véhiculer la masse, comme ce sera le cas dans le schéma a, plutôt que l'alimentation + 5 volts, comme ce sera le cas dans le schéma b. Un court-circuit éventuel entre les fils de liaison et la masse (boîtier métallique de l'appareil par exemple) est en effet sans danger dans le cas a alors qu'il peut conduire à la destruction de l'alimentation dans le cas b.

Ceci étant, la gestion d'un tel contact est fort simple comme le montre l'exemple suivant qui allume la LED câblée à demeure sur le port 13 de l'Arduino lors de l'appui sur un poussoir câblé selon le schéma 4.1b sur le port 2.

Listing 4.1 – Lecture de l'état d'un contact.

```
// Définition des constantes
const int contact = 2;
const int LED =  13;

// Variable contenant l'état du contact
int etatContact = 0;
```

© Dunod – La photocopie non autorisée est un délit.

```
void setup() {
  // Définition de LED en sortie
  pinMode(LED, OUTPUT);
  // Définition de contact en entrée
  pinMode(contact, INPUT);
}

void loop(){
  // Lecture de l'état du contact
  etatContact = digitalRead(contact);
  // Si le contact est fermé, le niveau logique est haut
  if (etatContact == HIGH) {
    // On allume la LED
    digitalWrite(LED, HIGH);
  }
  else {
    // On éteint la LED
    digitalWrite(LED, LOW);
  }
}
```

L'analyse de ce listing est fort simple puisque, après avoir initialisé la patte 13 en sortie et la patte 2 en entrée dans la fonction `setup` du programme, on vient tout simplement lire l'état de cette même patte 2 et, selon qu'elle est au niveau haut ou bas, c'est-à-dire selon que le poussoir est actionné ou pas, on allume ou non la LED.

4.1.2 Comment vaincre les rebondissements ?

Comme vous le savez certainement, tous les commutateurs sont sujets à ce que l'on appelle des rebondissements, c'est-à-dire que la fermeture ou l'ouverture du contact ne se fait pas immédiatement mais qu'elle est assortie d'une multitude d'oscillations parasites dont la durée peut être de quelques millisecondes. Au lieu de disposer aux bornes d'un tel commutateur qui s'ouvre du beau créneau visible figure 4.2a, on dispose alors du train d'impulsions parasites visible figure 4.2b.

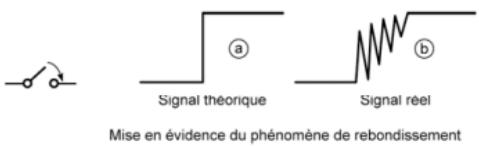

Figure 4.2 – Mise en évidence du phénomène de rebondissement.

Selon l'interprétation de l'état du bouton qui est faite par le logiciel, cela peut ou non avoir des conséquences fâcheuses. Il suffit pour vous en convaincre de reprendre le montage précédent et de tenter de le faire fonctionner avec le programme suivant qui ne gère pas les rebondissements.

Listing 4.2 – Mise en évidence de l'effet des rebondissements.

```
// Définition des constantes
const int contact = 2;
const int LED =  13;

// Variables contenant l'état du contact et de la LED
int etatContact = 0 ;
int etatLED = 0 ;

void setup() {
  // Définition de LED en sortie
  pinMode(LED, OUTPUT);
  // Définition de contact en entrée
  pinMode(contact, INPUT);
}

void loop(){
  // Lecture de l'état du contact
  etatContact = digitalRead(contact);
  // Si le contact est fermé, le niveau logique est haut
  if (etatContact == HIGH) {
    // On change l'état de la LED
    etatLED = 1 - etatLED ;
  }
  if (etatLED == 1) {
    // On allume la LED
    digitalWrite(LED, HIGH);
  }
  else {
    digitalWrite(LED, LOW);
  }
}
```

Vous allez constater que la LED s'allume et s'éteint, mais pas nécessairement après chaque appui sur le poussoir car, compte tenu de la vitesse d'exécution du programme d'une part et de la durée d'exécution des rebondissements d'autre part, l'état de la variable `etatContact` peut être testé plusieurs centaines de fois pendant la durée des rebondissements. De ce fait, la valeur de cette variable est quelconque au relâchement du bouton et la LED peut donc se trouver dans n'importe quel état.

En logique classique, on élimine les rebondissements avec des bascules RS, des monostables, ou des cellules RC devant des triggers de Schmitt. Ici c'est tout à fait inutile. Il suffit en effet que le logiciel qui vient lire la position des commutateurs fasse deux lectures successives séparées de quelques millisecondes pour s'affranchir de ce phénomène. On réalise ainsi un anti-rebondissement par logiciel.

Diverses variantes sont possibles selon que l'interrogation de l'entrée est systématique ou que l'on souhaite un fonctionnement sous interruption. Dans tous les cas, le secret de la fonction anti-rebondissement est une temporisation séparant deux lectures successives de la même information.

Toujours avec le même montage que celui que nous venons d'utiliser, à savoir la LED de la sortie 13 de la carte Arduino et le poussoir connecté à l'entrée 2, voici

© Dunod – La photocopie non autorisée est un délit.

un exemple de gestion des rebondissements qui permet, cette fois-ci, de bien faire changer la LED d'état à chaque action sur le poussoir.

Listing 4.3 – Exemple de traitement des rebondissements.

```
// Définition des constantes
const int contact = 2;
const int LED =  13;

// Variables contenant l'état du contact et de la LED
int etatContact = 0 ;
int etatPrecedent = 0 ;
int etatLED = 0 ;

void setup() {
  // Définition de LED en sortie
  pinMode(LED, OUTPUT);
  // Définition de contact en entrée
  pinMode(contact, INPUT);
}

void loop(){
  // Lecture de l'état du contact
  etatContact = digitalRead(contact);
  // Y a-t-il eu une transition ?
  if ((etatContact == HIGH) && (etatPrecedent == LOW)) {
    // On change l'état de la LED
    etatLED = 1 - etatLED ;
    delay(10) ; // Délai supérieur à celui des rebondissements
  }
  etatPrecedent = etatContact ;
  if (etatLED == 1) {
    // On allume la LED
    digitalWrite(LED, HIGH);
  }
  else {
    digitalWrite(LED, LOW);
  }
}
```

Après l'avoir essayé et constaté son bon fonctionnement, nous vous laissons le soin de l'analyser à la lecture des explications précédentes, ce qui ne devrait présenter aucune difficulté.

4.1.3 Claviers en matrices

De nombreuses applications ont besoin de disposer d'un clavier d'entrée d'informations, même si ce dernier n'a que peu de touches. Ainsi, un composeur téléphonique aura besoin de douze touches pour simuler un cadran de téléphone, un programmateur domestique aura besoin de touches d'entrée de l'heure et de la date, et ainsi de suite.

Lorsque le nombre de touches nécessaires est très faible, celles-ci peuvent être réalisées sous forme de poussoirs câblés comme expliqué ci-dessus, et le problème est résolu.

Lorsque le nombre de touches devient plus important, il faut adopter une autre solution. En effet, avec la méthode une touche – une patte d'entrée, on sature vite l'Arduino Uno et devoir faire appel à un Arduino Mega 2560 simplement pour disposer de plus d'entrées/sorties dans le seul but de connecter un clavier c'est un peu comme utiliser un marteau pilon pour écraser une mouche.

On adopte alors généralement la solution du clavier câblé en matrice, dont la figure 4.3 présente le principe dans le cas d'un clavier à douze touches, mais celui-ci est extensible à l'infini.

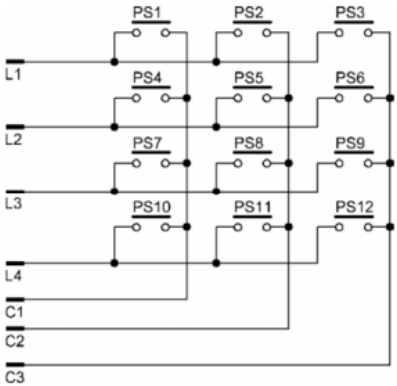

Figure 4.3 – Principe de câblage en matrice d'un clavier.

Les touches du clavier sont placées à l'intersection des lignes et des colonnes du quadrillage réalisé par les deux groupes de fils. L'appui sur une touche court-circuite donc une ligne avec une colonne. C'est cette situation qui est mise à profit pour localiser la position de la touche grâce à un logiciel de contrôle fort simple qui fonctionne de la façon suivante.

À l'instant *t*, les lignes sont placées en sorties et génèrent un niveau logique haut. Les colonnes quant à elles sont placées en entrées et vont donc pouvoir être lues.

La donnée émise sur les lignes, qui était initialement 1111, est remplacée par 1110 et la première colonne est lue. Si on y trouve un 0 logique on a identifié la touche actionnée qui se trouve à l'intersection de la première ligne et de la première colonne. Si ce n'est pas le cas, on lit la colonne suivante et ainsi de suite jusqu'à la dernière. Si aucun 0 logique n'a été trouvé, on applique alors 1101 sur les lignes et on recommence le processus précédent.

© Dunod – La photocopie non autorisée est un délit.

Lorsque la touche a été identifiée on dispose alors d'un code qui correspond à la position de cette dernière sur le clavier. Il ne reste donc plus qu'à le transcoder, au moyen d'un simple tableau, afin de le faire correspondre à l'indication figurant sur la touche, si bien sûr on souhaite disposer de cette information. Dans le cas contraire, ce code « brut » peut être utilisé tel quel pour valider la fonction déclenchée par la touche actionnée.

Même si l'écriture en C d'un tel programme reste relativement simple, puisque ce n'est qu'une suite d'entrées et de sorties parallèles agrémentées de quelques tests, la gestion d'un tel clavier est grandement facilitée par une bibliothèque spécialisée prévue à cet effet.

Cette bibliothèque n'est pas incluse dans l'installation standard de l'environnement de développement de l'Arduino et doit donc être téléchargée et installée manuellement. Elle se trouve à l'adresse *http://www.arduino.cc/playground/Code/Keypad* sous forme d'un seul et unique fichier compressé appelé keypad.zip.

Une fois téléchargé, ce fichier doit être décompressé dans le répertoire des bibliothèques de l'environnement de développement soit `\arduino-xx\librairies\` où cela provoque la création d'un répertoire appelé `keypad`. Si l'environnement de développement était déjà lancé lors de la mise en place de cette bibliothèque, il convient de le quitter et de le lancer à nouveau pour qu'elle soit prise en compte et apparaisse dans le menu **Croquis**, sous-menu **Importer bibliothèque**.

Cette bibliothèque est extrêmement puissante car elle autorise une définition très libre du clavier, du libellé de ses touches et de son mode de câblage. Voici un court exemple d'utilisation dans le cas du classique clavier « téléphonique » très répandu visible figure 4.4.

Figure 4.4 – Le clavier « téléphonique » très répandu utilisé en exemple.

Nous avons connecté ce clavier à l'Arduino comme indiqué figure 4.5 ; figure qui précise également à quelles intersections lignes – colonnes sont placées les diverses touches.

Dès lors, le programme suivant lit n'importe quelle touche actionnée sur ce clavier et envoie sa valeur via le port série sur le moniteur intégré à l'outil de développement.

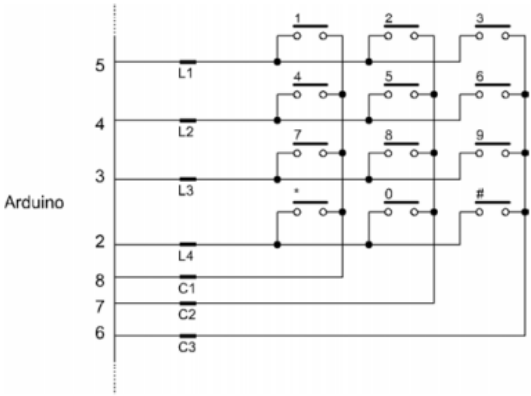

Figure 4.5 – Câblage et répartition des touches sur le clavier de la figure 4.4.

Listing 4.4 – Exemple de gestion d'un clavier en matrice.

```
#include <Keypad.h>

const byte ROWS = 4; // Définition du nombre de lignes
const byte COLS = 3; // Définition du nombre de colonnes
char keys[ROWS][COLS] = {
  {'1','2','3'},
  {'4','5','6'},
  {'7','8','9'},
  {'#','0','*'}
};
byte rowPins[ROWS] = {5, 4, 3, 2}; // Définition des ports
// où sont raccordées les lignes
byte colPins[COLS] = {8, 7, 6};    // Définition des ports
// où sont raccordées les colonnes

// Création du clavier
Keypad keypad = Keypad( makeKeymap(keys), rowPins, colPins, ROWS,
➥COLS );

void setup(){
  Serial.begin(9600); // Initialisation du port série
// à 9600 bits/seconde
}

void loop(){
  char key = keypad.getKey(); // Lecture de la touche actionnée

  if (key != NO_KEY){
    // Si une touche est actionnée, envoi sur le port série
    Serial.println(key);
  }
}
```

© Dunod – La photocopie non autorisée est un délit.

Après avoir appelé la bibliothèque `keypad` grâce à l'`include` de début, le programme définit la disposition des touches au moyen du tableau à deux dimensions `keys`. Les deux indices de ce tableau sont respectivement `ROWS` pour les lignes (ou rangées) et `COLS` pour les colonnes. Les tableaux unidimensionnels suivants définissent quant à eux le mode de câblage du clavier, c'est-à-dire les ports de l'Arduino auxquels sont respectivement raccordées les lignes et les colonnes.

La création du clavier a lieu avec la ligne `Keypad` qui permet à la fonction appropriée de la bibliothèque de prendre en compte les informations préalablement définies.

Pour lire le clavier, il ne reste plus alors qu'à appeler la fonction `keypad.getKey` qui restitue une variable de type `char` égale au code de la touche actionnée ou égale à `NO_KEY` si aucune touche n'a été actionnée. On peut difficilement imaginer plus simple. La bibliothèque permet également de définir un certain nombre de paramètres tels que la durée de rebondissement des touches, le temps pendant lequel l'utilisateur doit garder le doigt appuyé pour qu'une touche soit prise en compte, etc.

Ces fonctions d'usage plus rare sont décrites sur le site officiel consacré à l'Arduino sur la page dédiée à cette bibliothèque d'adresse : *http://www.arduino.cc/playground/Code/Keypad*

4.1.4 Plusieurs touches avec une seule entrée

Lorsque l'on ne doit gérer que trois ou quatre touches, aucune des solutions précédentes n'est réellement satisfaisante. La méthode simple une touche – une entrée gaspille inutilement des ports tandis que le clavier en matrice n'apporte que peu ou pas de bénéfice dans un tel cas. En effet, si l'on doit par exemple gérer cinq touches pour réaliser un pavé directionnel de navigation dans un menu (les quatre touches fléchées et la touche OK de validation) ; la solution une touche – une entrée va consommer cinq lignes de port, mais la solution du clavier en matrice ne sera pas meilleure car il faudra faire une matrice 2 × 3 ce qui consommera… cinq lignes de ports !

Il existe fort heureusement une autre solution, souvent méconnue, que l'on peut appeler le clavier analogique. La figure 4.6 présente son principe de réalisation, justement dans le cas des cinq touches que nous venons d'évoquer.

Comme vous pouvez le constater, cette solution est extraordinaire puisqu'elle ne consomme … qu'une entrée de l'Arduino ! Son principe de fonctionnement est fort simple. Les résistances R1 d'une part et R2 à R5 mises en série d'autre part forment un diviseur de tension ; tension qui est mesurée via une des entrées analogiques de l'Arduino.

Lorsque l'on appuie sur une des touches, on met en circuit une ou plusieurs des résistances R2 à R5 et on modifie donc le rapport de ce diviseur de tension et donc la valeur de la tension appliquée à l'entrée analogique de l'Arduino.

Pour exploiter un tel clavier, il suffit donc tout simplement de mesurer cette tension et de la comparer aux différentes valeurs possibles selon que telle ou telle résistance est en circuit et donc que telle ou telle touche est actionnée. C'est ce que fait le programme ci-dessous que vous pourrez directement transposer dans toute application utilisant un tel clavier.

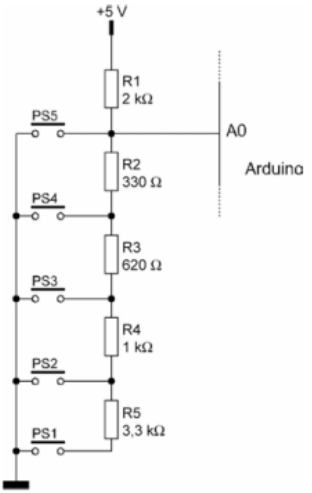

Figure 4.6 – Schéma de principe du clavier analogique.

Listing 4.5 – Exemple de gestion d'un clavier analogique.

```
int adc_key_val[5] ={
  30, 150, 360, 535, 760 };
int NUM_KEYS = 5;
int adc_key_in;
int key=-1;
int oldkey=-1;

void setup() {
}

void loop() {

  adc_key_in = analogRead(0);    // Lecture entrée analogique
  key = get_key(adc_key_in);     // Appel décodage touche
  if (key != oldkey)             // Si appui sur touche détecté
  {
    delay(50);                   // Délai d'anti-rebondissement
    adc_key_in = analogRead(0); // Lecture entrée analogique
    key = get_key(adc_key_in);  // Appel décodage touche
    if (key != oldkey)           // Touche détectée
    {
      oldkey = key;
      if (key >=0)
      {
        // Emplacement d'utilisation de la touche détectée
        // La variable key est égale à 0, 1, 2, 3 ou 4
```

© Dunod – La photocopie non autorisée est un délit.

```
        // Selon la touche qui a été actionnée
      }
    }
  }
}

// Conversion de la tension analogique
// en un numéro de touche

int get_key(unsigned int input)
{
  int k;
  for (k = 0; k < NUM_KEYS; k++)
  {
    if (input < adc_key_val[k])
    {
      return k;
    }
  }
  if (k >= NUM_KEYS)
  {
    k = -1;     // No valid key pressed
  }
  return k;
}
```

La première ligne de ce programme définit tout d'abord les différentes tensions qu'il sera possible d'obtenir selon celle des touches qui sera actionnée. Ces valeurs sont établies en tenant compte du fait que le convertisseur de l'Arduino est un modèle 10 bits, qui délivre donc une valeur maximum égale à 1023 pour une tension d'entrée de 5 volts.

Les différents seuils sont ensuite déterminés en calculant la tension délivrée par le pont diviseur selon celle des touches qui est actionnée. Le tableau 4.1 synthétise tout cela.

Tableau 4.1 – Détermination des différentes tensions délivrées par le clavier analogique.

Touche	Rapport de division	Tension de sortie	Donnée délivrée par le convertisseur
Aucune	1	5	1023
S1	0,724	3,62	741
S2	0,506	2,53	518
S3	0,322	1,61	329
S4	0,142	0,71	145
S5	0	0	0

Le programme principal qui exploite les données du clavier vient ensuite tout simplement lire la donnée délivrée par le convertisseur analogique/digital au moyen de la ligne `analogRead(0)` et appelle, avec cette donnée, la fonction `get_key`.

Cette fonction utilise une boucle `for` évoluant de 0 à 5 pour comparer la donnée délivrée par le convertisseur avec les cinq seuils définis précédemment. Compte tenu de l'ordre de test de ces seuils, dès que la donnée délivrée par le convertisseur devient inférieure à un seuil, la variable de boucle `k` correspond alors au numéro de la touche et c'est cette information qui est retournée par la fonction.

Par contre, si `k` a atteint sa valeur maximum de 5 sans que cette relation n'ait été satisfaite, cela signifie qu'aucune touche n'a été appuyée et la fonction retourne donc une valeur de `k` égale à –1.

Au niveau du programme principal, on constate que deux appels successifs à cette fonction sont réalisés avec un intervalle de temps de 50 ms entre les deux (à cause du `delay(50)`) et que les valeurs retournées par la fonction sont comparées. Si vous avez lu le § 4.1.2, vous aurez compris bien sûr qu'il s'agit tout simplement du traitement du rebondissement des touches.

Ce principe de clavier analogique peut bien sûr être étendu à plus de cinq touches. Il suffit pour cela d'adapter le schéma de la figure 4.6 et le listing précédent en conséquence. Il faut cependant veiller à ne pas l'utiliser avec un trop grand nombre de touches car plus celui-ci augmente, plus les fourchettes de tensions appliquées au convertisseur analogique/digital se resserrent et plus le montage devient sensible aux bruits parasites et aux dérives des valeurs des résistances utilisées. Une dizaine de touches nous semble être un maximum pour un tel clavier.

4.1.5 Entrées haute tension et opto-isolées

Lorsque l'application à base d'Arduino doit prendre en compte des informations disponibles sous haute tension ou des informations liées au secteur EDF par exemple, l'interfaçage direct est vivement déconseillé en raison des dangers d'électrocution qu'il comporte et la meilleure solution consiste alors à faire appel à un photocoupleur classique à diode ou transistor. Il suffit de le câbler comme indiqué figure 4.7.

Lorsque le circuit externe est sous tension, ce qui revient à dire qu'il passe du courant du côté LED du photocoupleur, le transistor qu'il contient est saturé et l'entrée de l'Arduino est vue comme étant au niveau logique bas (`LOW` en langage Arduino). Lorsque le circuit externe n'est pas alimenté, le transistor est bloqué et l'entrée est vue comme étant au niveau haut (`HIGH` en langage Arduino).

Le photocoupleur peut être n'importe quel modèle classique tel que, par exemple 4N25 ou 4N35. Le condensateur C1 est à mettre en place si l'on utilise ce montage pour détecter la présence ou l'absence de tension de manière statique ; pour voir si un appareil est alimenté par le secteur EDF par exemple.

Par contre, si l'on souhaite utiliser ce montage pour synchroniser l'Arduino avec la fréquence du secteur EDF, pour réaliser une horloge par exemple ou bien encore pour commander des gradateurs sans parasite, il ne faut pas mettre en place C1.

© Dunod – La photocopie non autorisée est un délit.

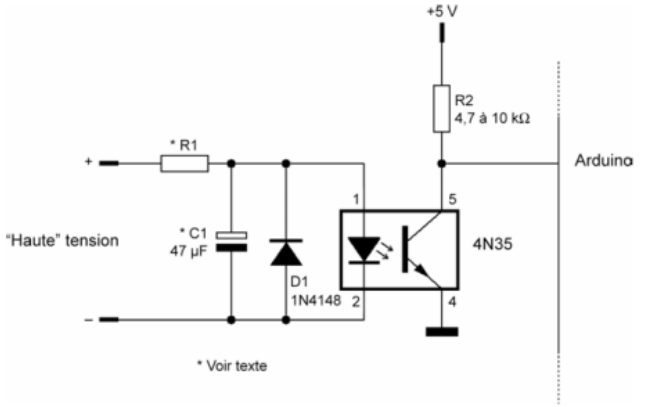

Figure 4.7 – Prise en compte d'information sous haute tension au moyen d'un photocoupleur.

On dispose alors aux bornes du transistor d'un signal qui est majoritairement au niveau logique bas et qui passe au niveau haut à chaque passage par zéro de la sinusoïde du secteur, soit exactement 100 fois par seconde.
La résistance R1 est à calculer de façon à ce que le courant traversant la LED du photocoupleur ne soit pas excessif. On peut utiliser pour cela la relation suivante :

$$R_1 = (V_{HT} - 1{,}7)/0{,}02$$

où V_{HT} est la valeur efficace de la haute tension.
Le processus de lecture de l'entrée de l'Arduino est identique à celui vu en début de chapitre pour un simple interrupteur. Seule la fonction anti-rebondissement peut être absente selon le circuit qui agit sur la LED du photocoupleur.

4.2 Les sorties numériques parallèles

Tout comme les entrées parallèles, les sorties parallèles sont très fréquemment utilisées dans les applications à base de microcontrôleurs. Elles sont ainsi mises à contribution pour allumer des LED, actionner des relais ou piloter des afficheurs, et encore si l'on s'en tient à leurs fonctions de base car, comme on peut les piloter individuellement de manière très souple, elles peuvent même être utilisées pour échanger des données sous forme série ! Elles sont généralement simples à mettre en œuvre car, comme les entrées, elles ne requièrent bien souvent que très peu de composants externes.

4.2.1 Commandes de LED et de relais

Nous commençons volontairement par le plus facile avec la commande de LED et de relais qui varie toutefois quelque peu compte tenu du courant que peuvent consommer de tels composants.
En effet, les ports numériques de l'Arduino sont des ports dits à fort courant (pour un microcontrôleur s'entend) car ils peuvent délivrer ou absorber jusqu'à 40 mA chacun. Des LED ordinaires peuvent alors être directement connectées sur ces ports avec interposition d'une simple résistance série de limitation de courant.

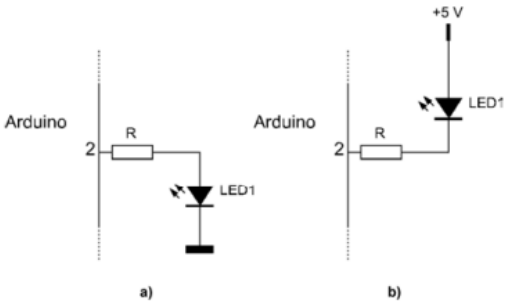

Figure 4.8 – Commande directe de LED par les ports à fort courant de l'Arduino.

La figure 4.8 montre ainsi à titre d'exemple la commande d'une LED par la sortie 2 de l'Arduino ; commande qui peut être réalisée comme indiqué en 4.8a, auquel cas la LED s'allumera pour un niveau logique haut en sortie, ou en 4.8b, auquel cas elle s'allumera pour un niveau logique bas en sortie.
Même si les ports de l'Arduino sont limités en courant et si l'on voit parfois sur Internet de telles LED câblées sans résistance série de limitation de courant, c'est une hérésie et fait courir un risque inutile au microcontrôleur qui équipe l'Arduino.
Une résistance série est donc obligatoire et se calcule très facilement de la façon suivante :

$$R = (5 - V_{LED})/I_{LED}$$

- V_{LED} est la tension de seuil de la LED (1,7 V pour le rouge et l'orange, 2 V pour le vert et 3,6 V pour le bleu et le blanc) ;
- I_{LED} est le courant traversant la LED (exprimé en A) ; courant qui ne doit en aucun cas excéder 40 mA pour ne pas surcharger les sorties de l'Arduino. La majorité des LED ordinaires se satisfait d'ailleurs d'un courant de l'ordre de 20 mA.

Si un affichage très lumineux est nécessaire, ou si l'on souhaite faire appel par exemple à des LED de puissance qui demandent nettement plus de 40 mA pour fonctionner correctement, il faudra utiliser le montage suivant qui consiste à intercaler sur les sorties de l'Arduino, un amplificateur à transistor, même rudimentaire.

© Dunod – La photocopie non autorisée est un délit.

La figure 4.9 en donne un exemple.

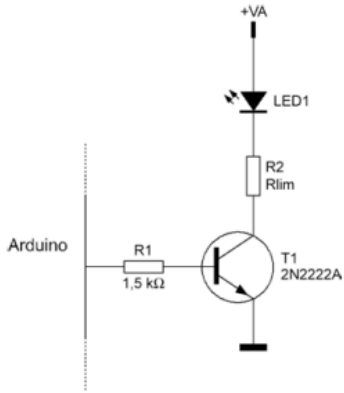

Figure 4.9 – Utilisation d'un transistor amplificateur pour disposer d'un courant de sortie plus important lors de la commande de LED de puissance.

La résistance R_{Lim} de limitation de courant de la LED est à calculer en fonction du courant désiré, comme indiqué ci-dessus, en remplaçant 5 par V_A qui est la valeur de la tension d'alimentation du montage ; tension qui n'est plus nécessairement égale à celle de l'Arduino compte tenu de la présence du transistor.

Compte tenu de la valeur de la résistance de base de ce transistor et de son gain, un courant pouvant atteindre 250 mA peut être fourni par ce montage, ce qui est généralement plus que suffisant pour des LED. Si tel n'est pas le cas, vous découvrirez dans la suite de ce chapitre comment commander des charges consommant plusieurs ampères à partir des sorties de l'Arduino.

La commande de photocoupleurs relève bien évidemment des mêmes méthodes que la commande de LED puisque l'organe d'entrée de ces composants est justement… une LED. Tout ce que nous venons de dire reste donc parfaitement valable étant entendu que les résistances de limitation de courant seront calculées compte tenu du courant maximum d'entrée des photocoupleurs utilisés.

Le principe de la commande de relais est très proche de ce que nous venons de voir pour les LED mais, compte tenu du courant important consommé même par les plus petits relais, il faut toujours faire appel à un transistor « de puissance » externe comme indiqué par exemple figure 4.10.

Comme les relais sont des composants inductifs, il ne faut surtout pas oublier la diode de protection montée en inverse sur la bobine. La tension d'alimentation de ce montage sera bien évidemment égale à la tension de collage du relais. Avec le transistor indiqué ici, tout relais nécessitant un courant inférieur ou égal à 200 mA pourra être utilisé, soit quasiment tous les relais habituellement rencontrés en électronique.

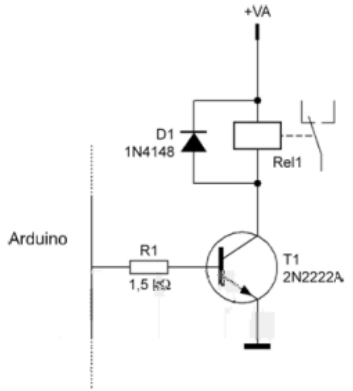

Figure 4.10 – Commande d'un relais au moyen d'un transistor externe.

Pour moins de quatre relais, on peut reproduire ce schéma en autant d'exemplaires que nécessaire mais, au-delà, il peut être plus intéressant d'utiliser un ULN 2803 qui, avec ses 500 mA de courant de sortie, peut commander jusqu'à 8 exemplaires de n'importe quel type de relais classique selon le schéma présenté figure 4.11.

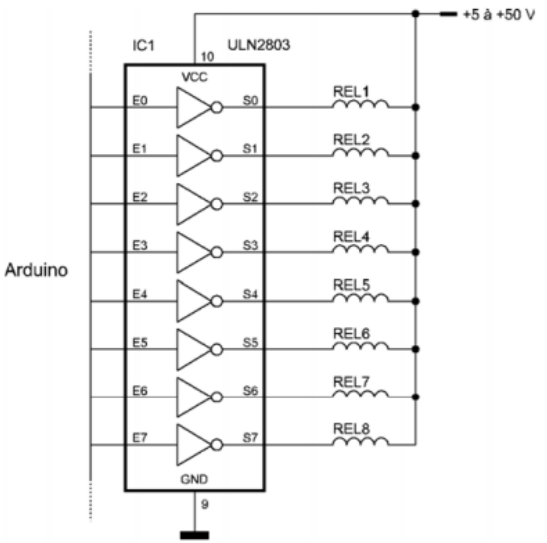

Figure 4.11 – Commande de plusieurs relais au moyen d'un ULN 2803.

© Dunod – La photocopie non autorisée est un délit.

Notez que la diode de protection visible figure 4.10 n'a pas besoin d'être prévue car elle est déjà intégrée dans le boîtier de l'ULN 2803 ce qui simplifie encore l'utilisation.

Le « logiciel », si l'on peut dire, nécessaire à la commande de toutes ces charges de sortie se résume à bien peu de chose puisqu'il suffit d'écrire :

```
void setup()
{
pinMode(sortie, OUTPUT) ;
}
```

Pour définir la ligne choisie en sortie, puis, dans le programme principal :

```
digitalWrite (sortie, HIGH) ;
```

ou

```
digitalWrite (sortie, LOW) ;
```

pour faire passer respectivement au niveau logique haut ou au niveau logique bas la sortie définie par la variable `sortie`, et donc pour faire allumer ou éteindre, coller ou décoller, la LED ou le relais qu'elle commande.

4.2.2 Commande directe de charges alimentées en continu

Même si c'est une pratique que nous déconseillons, tant en raison des problèmes d'isolement qui peuvent se poser, qu'en raison des parasites que cela peut induire sur l'alimentation de l'Arduino s'il s'agit de moteurs par exemple, voici néanmoins le principe à utiliser pour actionner directement des charges de forte puissance alimentées en courant continu.

Il suffit tout simplement de faire appel à un transistor Darlington de puissance, ou à un assemblage de transistors réalisant l'équivalent d'un Darlington. Ainsi, comme le montre la figure 4.12 réalisée avec un banal MJ 3001 (mais n'importe quel autre Darlington identique, plus ou moins puissant, peut être utilisé de la même façon), on peut commander, avec n'importe quelle sortie numérique de l'Arduino, des courants pouvant aller jusqu'à 10 ampères.

Si la ou les charges commandées sont inductives (relais et moteurs par exemple) il est primordial de ne pas oublier la diode de protection évoquée précédemment. De plus, pour les moteurs, il peut être nécessaire de prévoir une circuiterie typique d'antiparasitage avec selfs en série et condensateurs de découplage. Chaque cas étant un cas particulier dépendant des caractéristiques précises du moteur, il n'est pas possible de donner ici de schéma général.

Un tel circuit se commande bien évidemment comme les précédents par simple mise au niveau logique haut ou bas de la sortie correspondante de l'Arduino.

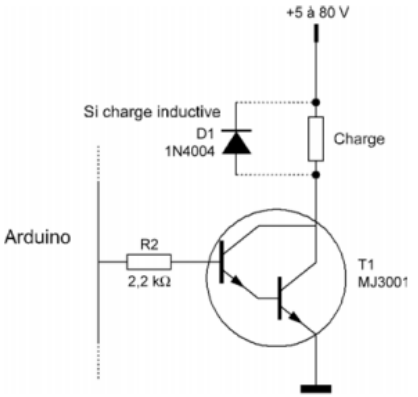

Figure 4.12 – Commande directe d'une charge de puissance alimentée en continu.

4.2.3 Commande directe de charges reliées au secteur

Les charges reliées au secteur peuvent évidemment être commandées par des relais qui, contrairement à ce que l'on lit encore trop souvent aujourd'hui, n'ont rien d'anachroniques à côté d'un microcontrôleur. Les relais modernes sont des composants fiables qu'il ne faut pas hésiter à utiliser lorsqu'aucune contrainte particulière ne s'y oppose (fortes vibrations pas exemple).

Lorsque l'on ne peut pas faire appel à des relais classiques ou relais électromécaniques, la seule solution à notre disposition est celle du relais statique, fort heureusement très facile à commander à partir des sorties de l'Arduino. En effet, que vous achetiez un relais statique du commerce ou que vous le réalisiez vous-même, il respecte, à de légères variantes près, le schéma de la figure 4.13 qui montre la commande d'une charge pouvant consommer jusqu'à 8 ampères grâce à un MOC 3040 ou 3041 ; photo triac très répandu aujourd'hui.

Ce circuit est un photo triac à détection de passage par zéro qui assure ainsi la mise en conduction du triac sans parasite, quel que soit le moment auquel l'Arduino déclenche la commande par rapport aux alternances du secteur. Bien sûr, vous pourriez réaliser la même chose avec un MOC 3020 ou 3021, qui ne possède pas de détection de passage par zéro, et réaliser celle-ci par logiciel. Dans le cas présent, la faible différence de prix entre un MOC 3021 et un MOC 3041 ne justifie toutefois pas à nos yeux le surcroît de travail nécessaire pour réaliser la détection de passage par zéro du secteur par logiciel.

Si vous préférez recourir à un relais statique du commerce, toute la partie encadrée de pointillés sur la figure 4.13 sera contenue dans ce dernier et son mode de connexion à l'Arduino ne changera donc pas.

Comme les précédents, un tel circuit se commande bien évidemment par simple mise au niveau logique haut ou bas de la sortie correspondante de l'Arduino.

© Dunod – La photocopie non autorisée est un délit.

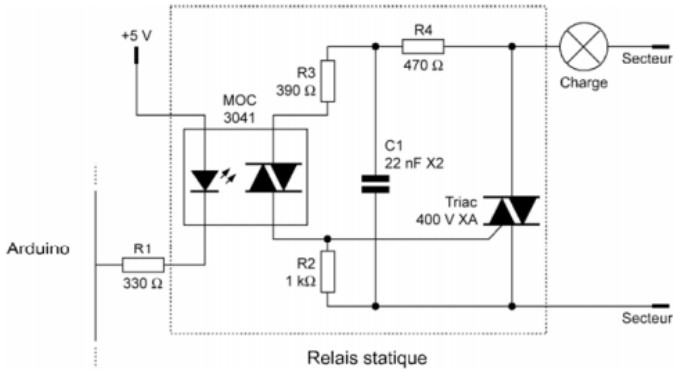

Figure 4.13 – Schéma d'un relais statique commandé par une sortie de l'Arduino.

4.2.4 Commande directe d'afficheurs à LED

La commande d'afficheurs à LED utilise a priori les mêmes principes que ceux vus ci-dessus pour les LED individuelles puisque des afficheurs de ce type ne sont rien d'autre que des groupements de LED de formes adéquates dans un même boîtier. Il faut cependant considérer deux cas selon le nombre d'afficheurs que l'on souhaite commander.

Si l'on n'a qu'un afficheur à piloter et si l'on dispose d'un nombre de lignes d'entrées/sorties suffisant, on peut faire appel au schéma de la figure 4.14. Les segments des afficheurs sont commandés directement par les sorties de l'Arduino, puisque ce ne sont rien d'autre que des LED et que le courant maximum qui peut être fourni (40 mA rappelons-le) est largement suffisant, même pour les afficheurs de grande dimension.

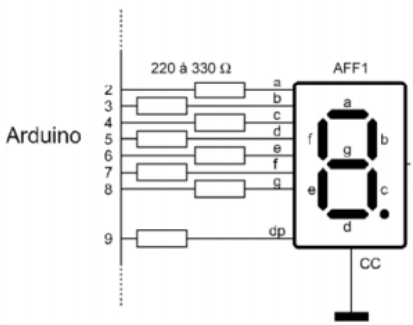

Figure 4.14 – Commande directe d'un afficheur sept segments à LED.

Un tel afficheur n'étant constitué que de LED, on peut très bien allumer ou éteindre ses segments un à un pour dessiner les chiffres ou symboles à afficher mais cela devient vite fastidieux, surtout si l'opération doit être répétée à de multiples reprises dans un programme. Il est donc préférable d'utiliser un tableau de codage précisant une fois pour toutes, pour chaque chiffre ou symbole à afficher, quels sont les segments à allumer. Le programme suivant présente un exemple de ce qu'il est possible de faire.

Listing 4.6 – Exemple de commande directe d'un afficheur sept segments.

```
// Définition du câblage de l'afficheur
// sur les sorties de l'Arduino

int segmentPins[] = {2, 3, 4, 5, 6, 7, 8, 9};

byte digits[10][8] = {
//  a  b  c  d  e  f  g  .
  { 1, 1, 1, 1, 1, 1, 0, 0},  // 0
  { 0, 1, 1, 0, 0, 0, 0, 0},  // 1
  { 1, 1, 0, 1, 1, 0, 1, 0},  // 2
  { 1, 1, 1, 1, 0, 0, 1, 0},  // 3
  { 0, 1, 1, 0, 0, 1, 1, 0},  // 4
  { 1, 0, 1, 1, 0, 1, 1, 0},  // 5
  { 1, 0, 1, 1, 1, 1, 1, 0},  // 6
  { 1, 1, 1, 0, 0, 0, 0, 0},  // 7
  { 1, 1, 1, 1, 1, 1, 1, 0},  // 8
  { 1, 1, 1, 1, 0, 1, 1, 0}   // 9
};

void setup()
{
  for (int i=0; i < 8; i++)
  {
    pinMode(segmentPins[i], OUTPUT);
  }
}

void loop()
{
int n = 4 ; // Affichage du chiffre 4 à titre d'exemple
  for (int i=0; i < 8; i++)
  {
    digitalWrite(segmentPins[i], digits[n][i]);
  }
}
```

On commence tout d'abord par définir comment sont reliés les segments de l'afficheur aux sorties de l'Arduino au moyen du tableau `segmentPins`. Les segments y sont placés de a à f pour finir par le point décimal.

Le tableau à deux dimensions, baptisé `digits`, contient quant à lui sur chaque ligne la liste de segments à allumer (représentés par un 1) et des segments à éteindre

© Dunod – La photocopie non autorisée est un délit.

(représentés par un 0) pour chaque chiffre ou symbole à afficher. Nous nous sommes limités ici aux chiffres de 0 à 9.
La fonction `setup` initialise ensuite toutes les lignes utilisées en sorties. Comme il y en a huit, nous réalisons cela avec une boucle `for`.
Pour afficher n'importe quel chiffre, il ne reste plus alors qu'à l'affecter à la variable `n` (rendue égale à 4 dans notre exemple) et à appeler la boucle qui fait suite. Celle-ci balaye la ligne du tableau `digits` correspondant à la valeur de `n` et envoie donc à l'afficheur des niveaux logiques hauts pour les 1 qui y sont lus et bas pour les 0, dessinant ainsi le chiffre correspondant.

4.2.5 Commande directe d'afficheurs à LED multiplexés

La méthode précédente fonctionne très bien mais présente l'inconvénient de consommer 7 ou 8 sorties de l'Arduino (7 si l'on n'utilise que les chiffres et 8 si l'on utilise les chiffres et le point décimal) par afficheur. S'il n'y en a qu'un, cela reste acceptable mais, dès que l'on veut en commander deux, et à plus forte raison plus de deux, cela n'est plus envisageable.
Dans une telle situation, il faut recourir à la technique du multiplexage qui fonctionne comme indiqué figure 4.15, dessinée dans le cas de deux afficheurs seulement mais extensible à l'infini.

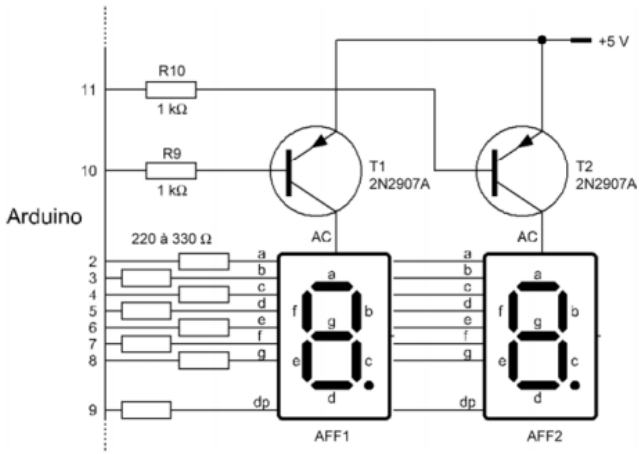

Figure 4.15 – Schéma de principe d'un affichage 7 segments à LED multiplexé.

Les segments des afficheurs sont toujours commandés par les 7 ou 8 sorties de l'Arduino comme nous venons de le voir, tandis que les anodes communes des afficheurs (dans cet exemple) sont validées à tour de rôle par des transistors commandés par deux autres sorties. Des afficheurs à cathodes communes peuvent évidemment

être utilisés en remplaçant les transistors PNP par des NPN et en reliant leurs émetteurs communs non plus au positif de l'alimentation mais à la masse.

Le principe de fonctionnement du programme de pilotage d'un tel affichage est fort simple. On présente sur les sorties segments le mot binaire correspondant aux segments à allumer et on valide ensuite la sortie correspondant au premier chiffre. Après une temporisation dépendant de la fréquence d'appel de ce programme, on invalide le premier chiffre et on reprend la même démarche pour le chiffre suivant. Les afficheurs sont donc allumés l'un après l'autre mais donnent l'illusion d'un affichage simultané en raison de la persistance des impressions rétiniennes. Cette dernière ayant une durée de l'ordre de 20 ms, il faut évidemment que l'allumage de chaque afficheur se répète suffisamment vite pour que l'œil n'ait pas une impression de papillotement.

Le programme de gestion d'un tel affichage reste relativement simple, surtout si l'on utilise comme base de départ celui que nous venons tout juste de réaliser pour un seul afficheur. En voici un exemple ci-dessous.

Listing 4.7 – Exemple de gestion de deux afficheurs sept segments en mode multiplexé.

```
// Définition du câblage des segments
// sur les sorties de l'Arduino

int segmentPins[] = {2, 3, 4, 5, 6, 7, 8, 9};

// Définition du câblage des transistors de commande
// des afficheurs sur les sorties de l'Arduino

int displayPins[] = {10, 11};

byte digits[10][8] = {
//  a  b  c  d  e  f  g  .
  { 1, 1, 1, 1, 1, 1, 0, 0},  // 0
  { 0, 1, 1, 0, 0, 0, 0, 0},  // 1
  { 1, 1, 0, 1, 1, 0, 1, 0},  // 2
  { 1, 1, 1, 1, 0, 0, 1, 0},  // 3
  { 0, 1, 1, 0, 0, 1, 1, 0},  // 4
  { 1, 0, 1, 1, 0, 1, 1, 0},  // 5
  { 1, 0, 1, 1, 1, 1, 1, 0},  // 6
  { 1, 1, 1, 0, 0, 0, 0, 0},  // 7
  { 1, 1, 1, 1, 1, 1, 1, 0},  // 8
  { 1, 1, 1, 1, 0, 1, 1, 0}   // 9
};

void setup()
{
  for (int i=0; i < 8; i++)
  {
    pinMode(segmentPins[i], OUTPUT);
  }
  pinMode(displayPins[0], OUTPUT);
  pinMode(displayPins[1], OUTPUT);
}
```

© Dunod – La photocopie non autorisée est un délit.

```
void loop()
{
  static int chiffre1 = 3 ;  // A titre d'exemple
  static int chiffre2 = 5 ;  // A titre d'exemple
  updateDisplay(chiffre1, chiffre2);
}

// Fonction de gestion du multiplexage

void updateDisplay(int value1, int value2)
{
  digitalWrite(displayPins[0], LOW);  // Allumage 1er afficheur
  digitalWrite(displayPins[1], HIGH); // Extinction 2ème afficheur
  setSegments(value1);
  delay(5);
  digitalWrite(displayPins[0], HIGH); // Extinction 1er afficheur
  digitalWrite(displayPins[1], LOW);  // Allumage 2ème afficheur
  setSegments(value2);
  delay(5);
}

// Fonction de gestion des segments

void setSegments(int n)
{
  for (int i=0; i < 8; i++)
  {
    digitalWrite(segmentPins[i], ! digits[n][i]);
  }
}
```

On retrouve dans ce programme le même principe de définition de connexion des segments que dans le listing 4.6 auquel on a adjoint la définition des sorties auxquelles sont reliés les transistors de commande.

Le tableau à deux dimensions de définition des segments à allumer en fonction du chiffre à afficher est également présent sous la même forme que précédemment puisque rien n'est modifié à ce niveau.

Par contre, l'allumage des segments par lecture de ce tableau est réalisé cette fois-ci sous forme de la fonction `setSegments` qui contient exactement la même chose que ce que nous avons utilisé au listing 4.6. Le recours à une fonction est ici nécessaire car elle va devoir être appelée lors de l'allumage de chaque afficheur.

La partie multiplexage proprement dite est confiée à la fonction `updateDisplay` qui admet en entrée deux données qui ne sont autres que les chiffres à afficher. Cette fonction réalise tout simplement ce que nous avons décrit ci-avant, à savoir l'allumage du premier chiffre, l'extinction du second et la présentation du code du premier chiffre sur les sorties à destination des segments puis, au bout d'un certain délai fixé ici à 5 ms, l'allumage du second chiffre, l'extinction du premier et la présentation du code du second chiffre sur les sorties segments suivie d'un nouveau délai de 5 ms.

Si l'on devait piloter par ce moyen plus de deux chiffres, il suffirait tout simplement d'ajouter la définition de la connexion de leurs transistors de commande au niveau de `displayPins` et, dans la fonction `updateDisplay`, d'ajouter l'extinction et l'allumage de ces afficheurs supplémentaires de la même façon que c'est fait ici pour le 1er et le 2e afficheur.

> **Attention !**
> Afin de respecter le délai maximum de persistance des impressions rétiniennes, la fonction `updateDisplay` doit être appelée régulièrement et à une fréquence suffisante dans le programme principal. C'est évidemment le cas ici puisque l'exemple ne fait rien d'autre qu'afficher la valeur des variables `chiffre1` et `chiffre2` mais cela peut parfois poser problème lorsque l'exécution d'autres fonctions du programme principal prend du temps.

4.2.6 Commande d'afficheurs à LED avec des circuits spécialisés

Même en utilisant la technique de l'affichage multiplexé, la gestion d'un afficheur sept segments à LED consomme un nombre non négligeable de sorties de l'Arduino puisque, dans tous les cas, il en faut au minimum 7 ou 8 pour les segments (selon que l'on utilise ou non la virgule) et une de plus par afficheur.

Dans certaines applications cela n'est pas acceptable compte tenu du nombre d'entrées/sorties numériques utilisées par ailleurs. Dans une telle situation, il ne reste plus alors que deux solutions : le recours à l'Arduino Mega 2560, nettement mieux doté en entrées/sorties numériques que le Uno, ou l'utilisation d'un circuit externe spécialisé.

Dans le cas d'un seul afficheur, ce circuit peut être un simple registre à décalage alors que, dans le cas de plusieurs afficheurs et donc du nécessaire recours au multiplexage, ce peut être un contrôleur spécialisé. Grâce à l'utilisation pour les piloter d'une liaison série synchrone, il ne faut plus alors que trois fils pour commander un afficheur pouvant comporter jusqu'à 8 chiffres comme nous le verrons au chapitre 6. En effet, l'utilisation de ces circuits mettant en œuvre, comme nous venons de le dire, une liaison série synchrone, nous avons estimé plus logique de vous la présenter au chapitre 6, consacré aux différents types de liaisons séries utilisables avec l'Arduino.

4.2.7 Utilisation d'afficheurs à cristaux liquides (LCD)

Lorsque l'application a des besoins conversationnels qui ne peuvent plus se satisfaire d'un banal affichage de type chiffres ou symboles simples, on fait très souvent appel aux afficheurs alphanumériques à cristaux liquides. On trouve en effet aujourd'hui à bas prix sur le marché des modèles autorisant l'affichage de deux ou quatre lignes de seize à quarante caractères. Qui plus est, ces caractères ne sont plus seulement limités à des chiffres mais à tous les caractères alphanumériques que l'on rencontre sur un clavier de micro-ordinateur.

Ces afficheurs sont toujours fournis sous forme d'un petit circuit imprimé support comprenant tout à la fois l'afficheur lui-même et l'électronique de gestion. Cette

© Dunod – La photocopie non autorisée est un délit.

électronique facilite la commande de l'afficheur par le microcontrôleur en le déchargeant de toute la partie gestion de l'afficheur et surtout de la génération, tout de même assez complexe, des signaux nécessaires.

Les miracles de la « standardisation automatique » dont sont coutumiers les fabricants d'Extrême-Orient font que quasiment tous ces afficheurs utilisent les mêmes contrôleurs, ou des contrôleurs compatibles. Ils s'interfacent donc tous avec les mêmes signaux, qui se gèrent de la même façon et ils comprennent tous un noyau d'ordres de base communs, même si certains modèles plus évolués que d'autres disposent de quelques commandes ou fonctions supplémentaires.

Interfacer et gérer un tel afficheur avec un Arduino reste une opération très simple comme nous allons le découvrir maintenant grâce aux schémas des figures 4.16 et 4.17.

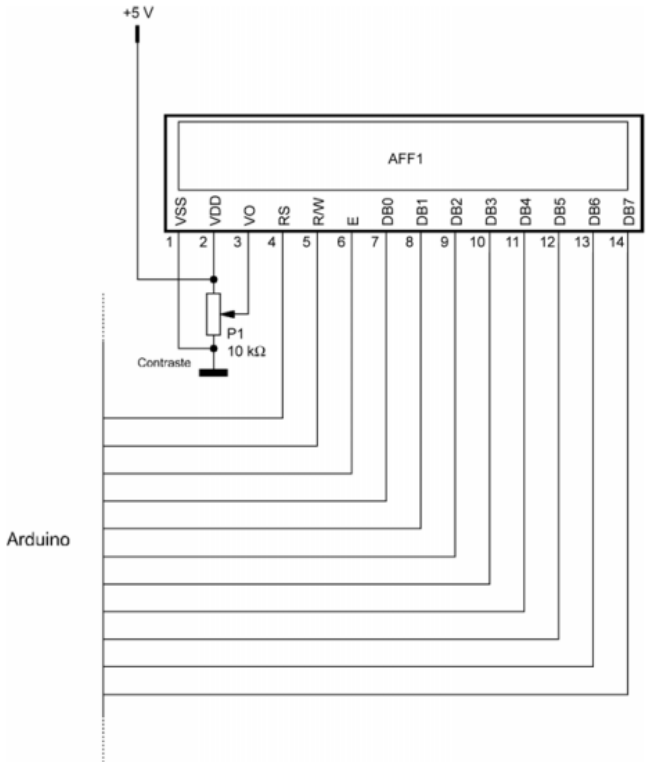

Figure 4.16 – Commande d'un afficheur LCD en mode 8 bits.

Comme vous pouvez le constater, les données échangées avec l'afficheur peuvent être codées sur 8 bits, via les lignes appelées DB0 à DB7, que l'on pilote alors à partir de huit sorties numériques de l'Arduino dans le cas de la figure 4.16. On dit alors tout naturellement que l'afficheur fonctionne en mode 8 bits.

Mais tous ces afficheurs peuvent aussi fonctionner comme indiqué figure 4.17, qui correspond à ce que l'on appelle alors le mode 4 bits. Dans un tel mode, les données sont transmises en deux fois sur 4 bits, ce qui consomme bien évidemment moitié moins des précieuses sorties numériques de l'Arduino.

Outre ces lignes de données, qui véhiculent tout à la fois les informations affichées et les commandes envoyées à l'afficheur, trois lignes de contrôle sont également nécessaires à son bon fonctionnement.

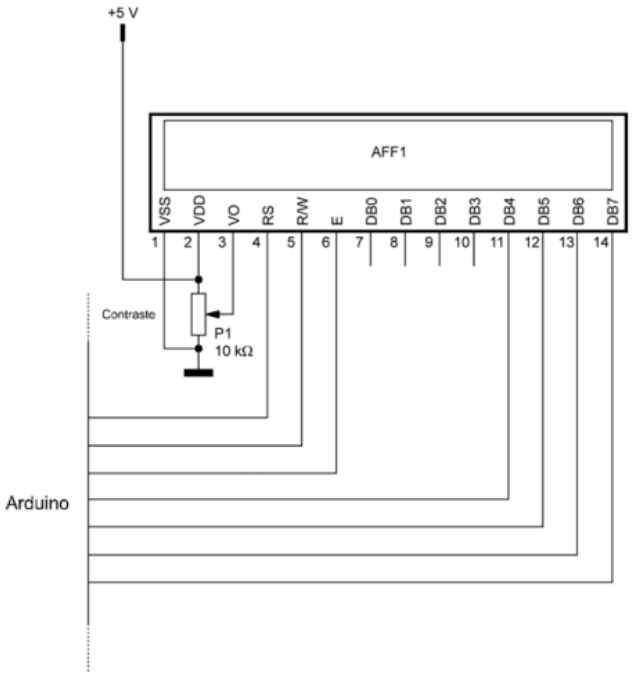

Figure 4.17 – Commande d'un afficheur LCD en mode 4 bits.

© Dunod – La photocopie non autorisée est un délit.

Ces lignes de contrôle ont les fonctions suivantes, valables pour tous les afficheurs de ce type :

– E ou *Enable* valide l'afficheur lorsqu'elle est au niveau haut. Il est alors à même de recevoir des commandes ou des caractères à afficher via ses lignes de données. Il reste insensible à leur état dans le cas contraire.
– R/W barre pour *Read/Write* barre indique si l'on veut écrire une donnée dans l'afficheur (R/W à zéro) ou lire ses informations (R/W à un). Il dispose en effet d'un registre interne capable de fournir certaines indications d'état. Lorsque l'on souhaite économiser des entrées/sorties, cette ligne est fréquemment mise à la masse, bloquant alors l'afficheur en mode écriture qui est son mode d'utilisation principal. Il devient alors impossible, dans ce cas, de lire ses registres d'état et sa mémoire de caractères mais cela n'est pas nécessairement contraignant dans la majorité des modes d'utilisation.
– RS pour *Register Select* indique si les lignes de données véhiculent des informations à afficher (RS à 1) ou des commandes de gestion de l'afficheur (RS à 0).

En mode informations à afficher, c'est-à-dire lorsque RS est au niveau logique 1, l'afficheur affiche successivement les caractères de code ASCII correspondant aux données reçues les uns à la suite des autres ; l'avancement de son curseur étant automatique.

Par ailleurs, un certain nombre de commandes permettent une gestion performante de cet affichage : déplacement du curseur de droite à gauche ou l'inverse, avec ou sans effacement de caractère, effacement de tout l'affichage, etc. Ces commandes sont en fait des codes qui doivent être envoyés sur les lignes de données de l'afficheur après avoir mis celui-ci en mode commande (RS à 0). Certains modèles disposent même d'une mémoire de génération de caractères interne dans laquelle vous pouvez stocker les formes de caractères de votre choix.

Le dialogue avec un tel afficheur est fort simple et se passe de la façon suivante dans le cas d'une écriture par exemple :

– mise à zéro de la ligne R/W ;
– positionnement de la ligne RS au niveau désiré selon que l'on souhaite envoyer un caractère à afficher ou une commande ;
– positionnement du code du caractère ou de la commande sur DB0 à DB7 (mode 8 bits) ou DB4 à DB7 (mode 4 bits) ;
– mise à un de la ligne E permettant la prise en compte de ces informations ;
– mise à zéro de la ligne E pour rendre à nouveau l'afficheur insensible à l'état de DB0 à DB7.

Ce processus peut se répéter autant de fois que nécessaire mais, compte tenu de la relative lenteur des afficheurs LCD, un délai d'attente doit être respecté entre l'envoi de deux données ou commandes successives. Sa valeur typique peut aller de 100 µs pour un simple affichage de caractère jusqu'à 2 millisecondes pour les opérations les plus complexes telles que l'effacement complet de l'afficheur par exemple. Une commande de lecture d'état est disponible si nécessaire mais n'est que rarement utilisée.

La réalisation du dialogue avec un tel afficheur ne présente pas de difficultés particulières puisqu'elle se résume à lire ou écrire des données binaires sur les ports parallèles reliés à l'afficheur, comme nous l'avons fait jusqu'à présent pour commander des relais ou des LED. Cela se révèle toutefois assez fastidieux à la longue, surtout si l'on veut pouvoir exploiter toutes les possibilités offertes par l'afficheur. Fort heureusement, il existe une bibliothèque spécialisée qui, d'externe qu'elle était jusqu'à présent, est aujourd'hui installée d'origine avec l'environnement de développement de l'Arduino.

Cette bibliothèque, baptisée comme il se doit `LiquidCrystal.h`, est très complète et dispose de 20 fonctions permettant une gestion très précise de n'importe quel afficheur LCD. Nous n'allons pas étudier ici en détail toutes ses fonctions car cela serait fastidieux et sans grand intérêt, vous renvoyant pour cela à leur page de description complète sur le site officiel consacré à l'Arduino d'adresse : *http://arduino.cc/en/Reference/LiquidCrystal*. Nous allons nous limiter ci-dessous aux fonctions principales, suivies d'un petit exemple de commande d'un tel afficheur.

■ LiquidCrystal()

Cette fonction crée une variable (en fait un objet pour parler correctement) de type `LiquidCrystal` et définit le mode de connexion de l'afficheur à l'Arduino sous l'une des formes suivantes :

```
LiquidCrystal(rs, enable, d4, d5, d6, d7) ;
LiquidCrystal(rs, rw, enable, d4, d5, d6, d7) ;
LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7) ;
LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7) ;
```

Elle supporte en effet tous les modes de connexion de l'afficheur à savoir : 4 bits avec mise à la masse de la ligne R/W, 4 bits avec utilisation de la ligne R/W, 8 bits avec mise à la masse de la ligne R/W et enfin 8 bits avec utilisation de la ligne R/W. Pour utiliser cette fonction, il suffit donc d'indiquer sur quelles entrées/sorties de l'Arduino est connecté l'afficheur en écrivant par exemple :

```
LiquidCrystal lcd(8, 9, 4, 5, 6, 7) ;
```

Cette ligne « baptise » notre objet de type `LiquidCrystal` du nom de `lcd` (mais vous pouvez choisir tout autre nom plus à votre goût !) et précise le mode d'utilisation de l'afficheur. Nous avons choisi ici une utilisation en mode 4 bits avec R/W mise à la masse et un câblage similaire à celui qui est réalisé sur le shield LCD – Keypad Module présenté au chapitre 3 (voir figure 3.13 si nécessaire).

■ begin()

Cette fonction, à appeler idéalement dans la fonction `setup` une seule fois avant toute utilisation de l'afficheur, permet de définir sa taille, c'est-à-dire le nombre de lignes et de caractères par ligne qu'il comporte. Elle s'utilise de la façon suivante :

```
lcd.begin(nombre de caractères, nombre de lignes) ;
```

© Dunod – La photocopie non autorisée est un délit.

Si vous avez donné à votre objet de type `LiquidCrystal` un autre nom que `lcd`, c'est évidemment ce dernier qui sera utilisé avant le point dans la ligne ci-dessus. Cette remarque étant valable pour tous les exemples qui vont suivre.

■ home() et clear()

Ces deux fonctions, d'usage très courant, permettent respectivement : de ramener le curseur en haut et à gauche de l'écran pour `home` sans toucher au contenu de l'affichage et de faire de même mais en effaçant l'affichage dans le cas de `clear`. Elles s'utilisent sous la forme :

```
lcd.home() ;
lcd.clear() ;
```

■ setCursor()

Comme son nom le laisse supposer, cette fonction positionne le curseur, c'est-à-dire définit l'emplacement qui va être utilisé pour afficher le caractère envoyé à l'afficheur suite à l'exécution de cette commande. Elle est évidemment très employée pour formater de manière aussi esthétique que possible les informations à afficher et s'utilise de la façon suivante :

```
lcd.setCursor (colonne, ligne) ;
```

où `ligne` et `colonne` indiquent la position choisie.

■ scrollDisplayLeft() et scrollDisplayRight()

Ces deux fonctions permettent de faire décaler l'affichage en cours et le curseur éventuellement associé d'une position vers la gauche (left), ou vers la droite (right). Leur utilisation est fort simple :

```
lcd.scrollDisplayLeft() ;
lcd.scrollDisplayRight() ;
```

■ autoscroll() et noAutoscroll()

La première de ces fonctions met en marche le mode « autoscroll » tandis que la seconde l'arrête. En mode « autoscroll », chaque affichage d'un caractère fait décaler automatiquement l'affichage d'une position vers la droite ou vers la gauche selon le sens de décalage préalablement défini (voir ci-dessous). De ce fait, chaque nouveau caractère affiché apparaît toujours à la même position sur l'affichage. L'utilisation est évidemment fort simple :

```
lcd.autoscroll() ;
lcd.noAutoscroll() ;
```

■ leftToRight() et rightToLeft()

Ces deux fonctions définissent le sens d'écriture sur l'afficheur. Par défaut, celui-ci est fixé de gauche à droite mais il peut être modifié par l'utilisation de l'une ou l'autre de ces fonctions. Comme indiqué ci-dessus, cela influe bien évidemment

sur le comportement de l'afficheur lorsqu'il est en mode « autoscroll ». Elles s'utilisent sous la forme :

```
lcd.leftToRight() ;
lcd.rightToLeft() ;
```

■ print()

Cette fonction est évidemment la plus utilisée puisque c'est celle qui fait effectivement afficher les caractères sur l'afficheur LCD. Les caractères apparaissent à partir de la position courante du curseur et se comportent comme l'ont éventuellement défini les commandes décrites ci-dessus.

En l'absence d'utilisation de ces dernières, ils apparaissent à partir de la position initiale du curseur avant l'appel de `print` et le curseur se décale automatiquement d'une position de gauche à droite pour chaque caractère affiché. La commande peut être utilisée sous deux formes :

```
lcd.print(donnée à afficher) ;
lcd.print(donnée à afficher, BASE) ;
```

`BASE` peut prendre l'une des valeurs suivantes : `BIN`, `DEC`, `OCT` ou `HEX` et est interprété comme nous l'avons vu au chapitre 2 lors de la description de la fonction `print` à laquelle nous vous prions de vous reporter si nécessaire.

Ceci étant vu, nous vous laissons le soin de découvrir si nécessaire les autres commandes à l'adresse indiquée ci-dessus car, d'une part elles sont d'usage moins courant que celles que nous venons de décrire, d'autre part cela ne présente aucune difficulté particulière.

Grâce à cette bibliothèque de gestion des afficheurs LCD, l'utilisation de ceux-ci devient d'une extrême simplicité comme le montre le court exemple suivant.

Listing 4.8 – Exemple d'utilisation d'un afficheur LCD en mode 4 bits.

```
#include <LiquidCrystal.h>

// Définition du mode de connexion de l'afficheur
// Ici en mode 4 bits avec R/W à la masse et le câblage
// utilisé sur le LCD-Keypad Module du chapitre 3

LiquidCrystal lcd(8, 9, 4, 5, 6, 7);

void setup() {
  // Définition du type d'afficheur
  // Ici un 2 lignes de 16 caractères
  lcd.begin(16, 2);
}

void loop() {
  // Positionnement du curseur ligne 0, colonne 5
  lcd.setCursor(5, 0);
  // Affichage du mot «Arduino»
  lcd.print(«Arduino»);
```

© Dunod – La photocopie non autorisée est un délit.

```
    // Positionnement du curseur ligne 1, colonne 5
    lcd.setCursor(5, 1);
    // Affichage du temps de fonctionnement du programme
    // exprimé en secondes
    lcd.print(millis()/1000);
}
```

Cet exemple se laisse très facilement analyser puisque, après la définition de l'objet baptisé LCD et tout à la fois de son mode d'utilisation et de son schéma de câblage grâce à la ligne `LiquidCrystal lcd(8, 9, 4, 5, 6, 7)` ; on définit son type dans la fonction `setup`.

La gestion de l'affichage proprement dit se résume ensuite à de simples appels des fonctions `setCursor` et `print`.

5 • ENTRÉES/SORTIES ANALOGIQUES : CAPTEURS, POTENTIOMÈTRES, GRADATEURS ET MOTEURS

Pendant des années les entrées/sorties analogiques ont été les parents pauvres des applications à base de microcontrôleurs car ces derniers n'intégraient que très rarement des convertisseurs analogiques/numériques et numériques/analogiques. Ce n'est plus le cas aujourd'hui, au moins dans une certaine mesure puisque, par exemple, l'Arduino dispose d'un convertisseur analogique/numérique de très bonne facture. Dans l'autre sens, c'est-à-dire dans le sens numérique/analogique, et même si nous avons vu au chapitre 2 l'existence d'une fonction `analogWrite`, c'est un peu moins vrai et les tensions analogiques restent générées au moyen de signaux numériques grâce à un artifice.

À l'image de ce que nous avons fait au chapitre précédent pour les interfaces numériques parallèles, nous vous proposons de découvrir maintenant comment utiliser ces entrées/sorties analogiques pour interfacer votre Arduino avec des éléments différents de ceux que nous avons manipulés jusqu'à présent.

5.1 Les entrées analogiques

Comme nous l'avons vu au chapitre 2, l'Arduino Uno dispose de six entrées analogiques, repérées A0 à A5, tandis que l'Arduino Mega 2560 en propose 16, repérées A0 à 15.

Ces entrées sont reliées à un convertisseur analogique/digital intégré dans le microcontrôleur AVR qui les équipe ; convertisseur qui fournit un résultat sur 10 bits c'est-à-dire compris entre 0 et 1023. Une tension d'entrée nulle lui fera donc délivrer en sortie une donnée numérique nulle tandis que lorsque sa tension d'entrée sera égale à sa tension de référence, il délivrera une donnée numérique égale à 1023. Par défaut, cette tension de référence est égale à la tension d'alimentation du microcontrôleur. Si donc la tension de référence est de 5 volts, le convertisseur délivrera 1023 lorsque sa tension d'entrée sera égale à 5 volts, ce qui lui conférera alors une résolution de 5/1024 soit environ 4,9 mV. Par contre, si cette tension de référence n'est plus que de 2 volts par exemple, la résolution deviendra meilleure puisqu'elle atteindra alors 2/1024 soit environ 1,9 mV.

© Dunod – La photocopie non autorisée est un délit.

Cette tension de référence peut provenir de l'extérieur ou bien être générée en interne par le microcontrôleur, comme nous l'avons vu au chapitre 2. Dans tous les cas, il est possible de la définir grâce à la fonction `analogReference` qui a été décrite dans ce même chapitre.

Ceci étant, le convertisseur analogique/digital contenu dans le microcontrôleur qui équipe les différentes versions d'Arduino reste un élément fragile qu'il est assez facile de détruire si l'on ne prend pas un minimum de précautions. Avant de passer à la pratique, nous vous rappelons donc ci-dessous les recommandations que nous avons déjà formulées au chapitre 2 à savoir que :

– la tension d'entrée, appliquée sur n'importe quelle entrée analogique (A0 à AX), ne doit en aucun cas dépasser la tension d'alimentation du microcontrôleur ;
– si une tension de référence externe appliquée à AREF est utilisée, il faut impérativement la définir au moyen de `analogReference(EXTERNAL)` avant d'appeler la fonction `analogRead`. Dans le cas contraire, un court-circuit pourrait avoir lieu entre la source de tension de référence interne et cette tension externe pouvant conduire à la destruction du microcontrôleur ;
– enfin, et même si cela n'a pas de conséquences fâcheuses, la tension d'entrée appliquée au convertisseur doit toujours être inférieure ou égale à la tension de référence utilisée. Celui-ci délivre en effet en sortie la donnée maximum permise par ses 10 bits, à savoir 1023, lorsque la tension d'entrée est égale à la tension de référence. Toute valeur supérieure correspond donc aussi à 1023 mais cela n'a alors plus de signification.

Rappelons également, avant de passer à la pratique, que le convertisseur analogique/digital a besoin de 100 µs environ pour réaliser une conversion complète. Il convient donc d'en tenir compte lors de l'utilisation d'`analogRead` et de se limiter à un maximum de 10 000 appels de cette fonction par seconde.

5.1.1 Utilisation de capteurs analogiques résistifs

Même si le numérique prend de plus en plus le pas sur l'analogique dans le monde des capteurs, un certain nombre d'entre eux délivrent encore leur information sous forme analogique. Cela peut être directement sous forme d'une tension, comme nous le verrons ci-après avec des capteurs de température par exemple, mais cela peut être également indirect compte tenu du phénomène physique mis en œuvre par le capteur.

Une photorésistance par exemple, appelée aussi LDR pour *Light Dependent Resistor* ou, dans les pays anglo-saxons, cellule CdS (abréviation de sulfure de cadmium qui est le principal matériau qu'elle contient) voit ainsi sa résistance varier en fonction de son éclairement. Une CTN, ou résistance à coefficient de température négatif, utilisée pour mesurer la température, est dans le même cas.

Elles ne vont donc pas délivrer directement une tension mais, comme on ne sait pas mesurer une résistance avec un convertisseur analogique/digital, on va transformer cette variation de résistance en variation de tension au moyen de l'un des deux schémas de la figure 5.1.

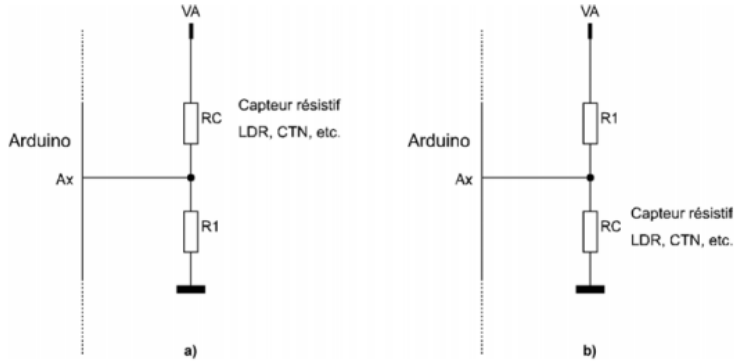

Figure 5.1 – Le classique diviseur de tension utilisé pour les capteurs résistifs.

Les deux montages se valent mais, comme nous l'avons fait pour la figure 4.1 du chapitre précédent, nous donnerons la préférence à celui où une extrémité du capteur est reliée à la masse si ce dernier doit être déporté de l'Arduino (revoir si nécessaire le justificatif que nous avons donné au niveau de cette même figure 4.1 à ce sujet).

La tension délivrée par ce montage est donnée par l'une des deux relations suivantes :

– $V_S = V_A \cdot R_1/(R_1 + R_C)$ dans le cas de la figure 5.1a ;
– $V_S = V_A \cdot R_C/(R_1 + R_C)$ dans le cas de la figure 5.1b ;

où V_A est la tension d'alimentation du pont diviseur (généralement les 5 volts de l'alimentation de l'Arduino) et où R_C est la résistance du capteur.

Il ne reste plus alors qu'à relier V_S à l'une des entrées analogiques de l'Arduino et à écrire :

```
tension = analogRead(entrée) ;
```

pour disposer dans la variable `tension` de la valeur de V_S appliquée sur `entrée`.

Cela fonctionne très bien si l'on souhaite, par exemple, déclencher un appareil lorsque l'éclairement dépasse un certain seuil, dans le cas d'une LDR, ou lorsqu'une température dépasse une certaine valeur, dans le cas d'une CTN. Il suffit en effet de mesurer avec un voltmètre la tension V_S obtenue lorsque ce seuil est atteint et de la reporter ensuite dans le programme.

Ainsi, le court exemple ci-dessous, que vous pourrez réaliser très facilement et sans aucun câblage au moyen du Danger Shield présenté au chapitre 3, montre comment réaliser un éclairage automatique reposant sur ce principe. Il fait passer la sortie numérique 5, à laquelle se trouve reliée une LED, au niveau haut, allumant ainsi cette dernière lorsque la LDR connectée comme indiqué figure 5.1b sur son entrée A3 est suffisamment dans l'obscurité.

© Dunod – La photocopie non autorisée est un délit.

Listing 5.1 – Exemple de prise en compte de la variation d'un capteur résistif.

```
int val = 0;

// Définition des entrées/sorties

#define LED1  5
#define LDR  3

void setup()
{
  pinMode(LED1, OUTPUT);
}

void loop()
{
  val = analogRead(LDR);  // Lecture de la tension d'entrée
  if (val < 512)          // Comparaison au seuil choisi
  {
    digitalWrite(LED1, HIGH) ;
  }
  else
  {
    digitalWrite(LED1,LOW) ;
  }
}
```

5.1.2 Où il est question d'hystérésis

Si vous réalisez le montage correspondant à l'exemple précédent, vous constaterez qu'il fonctionne … mal !

En effet, si la LDR est franchement dans l'ombre, la LED est bien allumée. Si elle est franchement à la lumière, la LED est bien éteinte mais, au voisinage du seuil de basculement, la LED adopte parfois une luminosité intermédiaire due au fait que la sortie de l'Arduino qui la commande passe sans cesse du niveau haut au niveau bas et vice versa.

La raison d'être de ce comportement est facile à comprendre. Notre programme compare en effet la valeur délivrée par le convertisseur avec un seul seuil, 512 dans notre exemple, pour décider ou non d'allumer la LED. Or, nous avons vu ci-avant que la résolution du convertisseur était de 4,9 mV. En d'autres termes, il suffit que V_S varie de 4,9 mV dans un sens ou dans l'autre pour que l'on franchisse ce seuil et que l'on passe donc d'allumé à éteint et vice versa. Un changement infime de luminosité suffit donc pour cela.

Pour éviter ce phénomène, tous les montages de ce type, qui exploitent l'état d'un capteur pour prendre une décision, utilisent ce que l'on appelle l'hystérésis, c'est-à-dire que le seuil de basculement dans un sens est différent du seuil de basculement dans l'autre sens. Si l'écart entre les deux est suffisant, on évite ainsi toute oscillation au voisinage de ces derniers.

Il suffit, pour vous en persuader, de manœuvrer doucement dans un sens puis dans l'autre le thermostat de votre chauffage ou de votre radiateur électrique. Vous constaterez alors que la position qui l'arrête est assez éloignée (en général de un ou deux degrés) de celle qui le remet en marche.
Pour introduire de l'hystérésis dans l'exemple précédent, il suffit de modifier le programme comme indiqué avec le listing que voici.

Listing 5.2 – Exemple de prise en compte de la variation d'un capteur résistif avec hystérésis.

```
int val = 0;

// Définition des entrées/sorties

#define LED1  5
#define LDR  3

void setup()
{
  pinMode(LED1, OUTPUT);
}

void loop()
{
  val = analogRead(LDR);   // Lecture de la tension d'entrée
  if (val < 512)           // Comparaison au seuil bas choisi
  {
    digitalWrite(LED1, HIGH) ;
  }
  else
  {
    if (val >620)         // Comparaison au seuil haut choisi
    {
    digitalWrite(LED1,LOW) ;
    }
  }
}
```

Le programme est majoritairement identique au précédent mais, cette fois-ci, on utilise deux seuils suffisamment éloignés l'un de l'autre (512 et 620 dans le cas présent) pour décider d'allumer ou d'éteindre la LED. De petites variations de luminosité, et donc de résistance du capteur, ne suffisent donc plus à provoquer une excursion de tension d'un seuil à l'autre et tout risque d'oscillation est donc éliminé.

5.1.3 Utilisation de capteurs délivrant une tension analogique

Si l'on souhaite réaliser de véritables mesures, et non seulement des prises de décision si l'on dépasse tel ou tel seuil, il est impossible de recourir au simple diviseur de tension de la figure 5.1. En effet, comme le montrent à l'évidence les relations entre la tension de sortie V_S et la valeur de la résistance du capteur R_C ; V_S n'est pas

© Dunod – La photocopie non autorisée est un délit.

une fonction linéaire de cette dernière. Donc, même si le capteur est linéaire, ce qui est en outre rarement le cas, la mesure réalisée ainsi ne le sera pas.

Bien sûr, il est possible, au moyen de tables par exemple, de compenser cette non-linéarité mais, et même si ces solutions ont pu être employées par le passé, elles sont aujourd'hui tombées en désuétude en raison de la mise sur le marché de capteurs parfaitement linéaires et délivrant de surcroît une tension directement mesurable.

Nous vous proposons ainsi à titre d'exemple de mesurer une température au moyen d'un circuit intégré capteur très répandu, le TMP 36 d'Analog Devices. Ce circuit, dont vous trouverez la fiche technique complète dans les compléments en ligne, ne nécessite aucun composant externe pour être utilisé et s'alimente sous une tension qui peut aller de 2,7 à 5,5 volts. Il peut mesurer toute température comprise entre – 40 °C et 125 °C avec une précision de +/– 1 °C à 25 °C et de +/– 2 °C sur toute l'étendue de mesure, et avec une linéarité meilleure que +/– 0,5 °C.

Cerise sur le gâteau, il délivre une tension de sortie qui varie de 10 mV par degré Celsius, ce qui facilite donc grandement la conversion tension-température. Comme il ne peut délivrer de tension négative, pour la mesure des températures inférieures à 0 °C, puisqu'il n'est alimenté que par une seule tension positive, sa tension de sortie est affectée d'un « offset », ou décalage en bon français, de 500 mV. Il délivre ainsi 500 mV pour 0°C et donc, par exemple, 750 mV pour 25 °C ou bien encore 300 mV pour – 20 °C.

L'utilisation d'un tel capteur avec un Arduino est d'une extrême simplicité, tant matérielle que logicielle. Le schéma de mise en œuvre est en effet réduit à ce que vous pouvez découvrir figure 5.2.

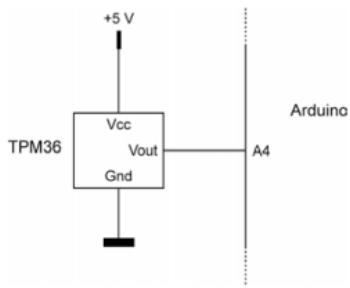

Figure 5.2 – Utilisation d'un capteur de température TMP36 d'Analog Devices.

Ce schéma, que vous pouvez réaliser sans aucun câblage avec le Danger Shield présenté au chapitre 3, permet de tester très facilement le programme suivant qui exploite l'information délivrée par le TMP 36 pour afficher la température en clair sur l'écran du PC relié à l'Arduino.

Listing 5.3 – Exemple de mesure de température au moyen d'un TMP 36 utilisant l'arithmétique en virgule flottante.

```
#define capteur 4
float temperature ;
float tension ;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  tension = (analogRead(capteur) * 4.9);
                                    // Conversion en tension
réelle
  temperature = (tension - 500)/10;  // Conversion en température
                                    // et prise en compte du
décalage
  Serial.print("Temperature : ");
  Serial.println(temperature);
  delay(1000);
}
```

Ce programme est tout naturellement ce qui vient à l'esprit dans une telle situation. Il commence par lire la valeur fournie par le convertisseur analogique/digital et convertit cette dernière en tension réelle, sachant en effet que la résolution de ce dernier est de 4,9 mV comme nous l'avons vu précédemment.

Il ne reste plus ensuite qu'à soustraire du résultat les 500 mV de tension de décalage et à diviser le résultat par 10 puisque le capteur délivre 10 mV par °C pour obtenir la valeur de la température.

Si vous essayez ce programme, vous constaterez qu'il fonctionne et qu'il affiche la température avec deux décimales… ce qui est un peu ridicule pour un capteur dont la précision est au mieux de +/– 1 °C !

En présence de capteurs aussi peu précis, et plutôt que de recourir systématiquement à des variables en virgule flottante qui sont très consommatrices de mémoire et de temps de calcul, il est également possible de faire appel à des fonctions moins connues du langage de l'Arduino telle que, par exemple, `map`. On peut alors écrire le programme suivant.

Listing 5.4 – Exemple de mesure de température au moyen d'un TMP 36 sans utiliser l'arithmétique en virgule flottante.

```
#define capteur 4
int tension ;
int temperature ;

void setup()
{
```

© Dunod – La photocopie non autorisée est un délit.

```
  Serial.begin(9600) ;
}

void loop()
{
  tension = map(analogRead(capteur),0,1023,0,500);
  temperature = (tension - 50) ;
  Serial.print(«Temperature : «) ;
  Serial.println(temperature) ;
  delay(1000) ;
}
```

Ce programme, qui n'occupe que 2980 octets en mémoire de l'Arduino contre 4156 pour le précédent, effectue le même travail et, même s'il ne donne un résultat que sous forme d'un nombre entier puisqu'il ne manipule que ce type de variable, est tout aussi exact que lui compte tenu de la précision du capteur.

L'essentiel de la conversion entre la donnée délivrée par le convertisseur et la tension réelle est réalisé par la seule fonction `map(analogRead(capteur), 0, 1023, 0, 500)` qui transforme automatiquement la plage 0-1023 délivrée par le convertisseur en plage 0-500, puisque nous voulons disposer ici d'une résolution de 10 mV compte tenu du fait que le capteur fournit 10 mV par °C.

En présence d'un capteur délivrant une tension proportionnelle au paramètre physique qu'il mesure, il ne faut donc pas hésiter à recourir à la fonction `map`, sous réserve bien sûr que la précision du capteur ne soit pas supérieure à ce que l'on est en droit d'attendre de cette fonction.

5.1.4 Utilisation de capteurs délivrant « n'importe quoi »

Sous ce titre quelque peu iconoclaste se cache une utilisation particulière, quoique pourtant bien utile, des entrées analogiques. En effet, dans un certain nombre de situations, on ne souhaite pas nécessairement mesurer précisément un paramètre physique au moyen d'un capteur mais seulement détecter sa présence, sans plus.

Un exemple concret de cette situation peut être fourni par la réalisation d'un système d'alarme anti-intrusion où l'on souhaite utiliser un détecteur de choc. Un tel détecteur peut être réalisé très facilement avec… un buzzer piézoélectrique.

En effet, un tel composant contient un disque de céramique piézoélectrique sur lequel sont fixées deux électrodes. Lorsqu'on leur applique une tension alternative, elles font vibrer le disque mais, réciproquement, lorsque l'on fait vibrer le disque, une tension d'autant plus grande que les vibrations sont importantes apparaît aux bornes des électrodes.

Il suffit alors de relier un tel buzzer à une entrée analogique d'un Arduino pour en faire un détecteur de chocs aussi efficace que simple. La figure 5.3 montre un exemple de schéma que vous pourrez tester sans le moindre câblage au moyen du Danger Shield présenté au chapitre 3.

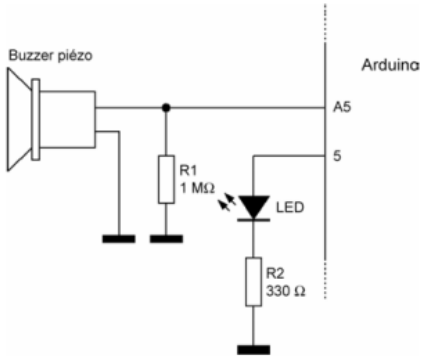

Figure 5.3 – Réalisation d'un détecteur de chocs avec un buzzer piézoélectrique.

L'exploitation de l'information délivrée par un tel capteur est fort simple comme le montre à titre d'exemple le listing suivant.

Listing 5.5 – Exemple d'utilisation d'un détecteur de chocs.

```
#define buzzer 5
#define LED 5

int chocs ;

void setup()
{
  pinMode(LED, OUTPUT);
}

void loop()
{
  chocs = analogRead(buzzer) ;
  if (chocs > 5)
  {
    digitalWrite(LED, HIGH) ;
    delay(2000) ;
    digitalWrite(LED, LOW) ;
  }
}
```

Le programme se passe quasiment de commentaire puisque, comme vous pouvez le constater, il se limite à la lecture de la valeur délivrée par le convertisseur analogique/digital et, dès que celle-ci dépasse un certain seuil (5 dans cet exemple), il allume une LED.

Nous attirons votre attention sur la faiblesse du seuil qui correspond ici à une tension réelle de l'ordre de 25 mV. En effet, la tension délivrée par un buzzer

© Dunod – La photocopie non autorisée est un délit.

piézoélectrique est très faible, surtout si l'on fait appel à un modèle encapsulé dans un boîtier dans lequel le disque piézo est souvent de petit diamètre. Sachez cependant qu'il existe de tels disques piézoélectriques de grand diamètre (jusqu'à 30 mm) vendus nus, mais néanmoins munis de leurs électrodes auxquelles sont soudés les fils de connexion. Ils s'avèrent évidemment nettement plus sensibles aux chocs qu'un buzzer dont ce n'est pas l'usage premier.

5.2 Les sorties analogiques PWM

Comme nous l'avons déjà dit à plusieurs reprises, l'Arduino ne dispose pas d'un véritable convertisseur digital/analogique, et ce bien que son langage propose la fonction `analogWrite` tendant à nous faire croire le contraire. Cette fonction exploite en effet la possibilité qu'ont les microcontrôleurs AVR de générer des signaux PWM, ce qui signifie Pulse Width Modulation ou, en français, modulation de largeur d'impulsions ou encore MLI.
Ces signaux permettent de doser avec une grande efficacité la puissance appliquée à des charges alimentées sous une tension continue et, sous certaines conditions, permettent également de disposer d'une tension analogique comme nous allons le voir dans la suite de ce chapitre.

5.2.1 Gradateur de tension ou de puissance PWM

Toutes les charges alimentées sous une tension continue voient leur puissance varier lorsque l'on fait varier leur tension d'alimentation ou, plus exactement, la valeur moyenne de cette dernière. C'est le cas par exemple des ampoules à incandescence mais aussi des moteurs électriques continus.

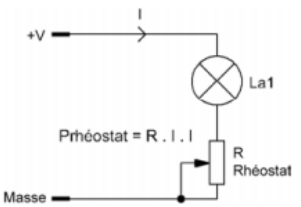

Figure 5.4 – Une méthode de gradation très peu performante.

La figure 5.4 montre comment est réalisée habituellement la gradation de l'éclairage du tableau de bord sur une voiture, choisie à titre d'exemple car elle est très représentative du problème. Un rhéostat, c'est-à-dire un potentiomètre de puissance bobiné, est placé tout simplement en série avec l'alimentation des ampoules. Cela fonctionne mais équivaut parfois à dissiper en chaleur dans le rhéostat autant de puissance que dans les ampoules. Ainsi par exemple, si le tableau de bord est

éclairé par deux ampoules de 5 watts et que vous ayez réglé la luminosité à la moitié de sa valeur maximale, les deux ampoules dissiperont à peu près 5 watts (10 watts divisés par 2) mais le rhéostat dissipera aussi 5 watts en pure perte sous forme de chaleur.

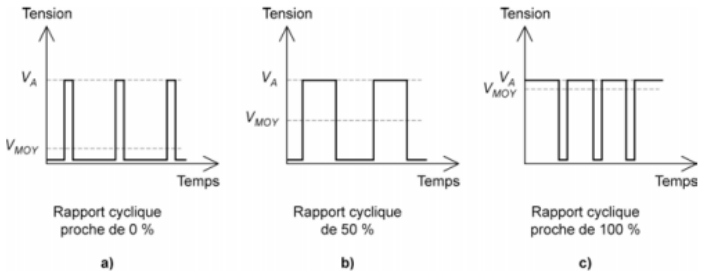

Figure 5.5 – Relation entre valeur moyenne et rapport cyclique de signaux PWM.

Une solution beaucoup plus performante consiste à faire appel à des signaux PWM qui sont constitués par des impulsions à fréquence fixe, d'amplitude égale à la tension d'alimentation totale, mais de rapport cyclique réglable en fonction de la luminosité désirée.

Lorsque les signaux ont un rapport cyclique très faible, comme schématisé en 5.5a, la valeur moyenne de la tension correspondante est aussi très faible et les ampoules rougeoient tout juste. Lorsque ce rapport cyclique est égal à 50 % comme schématisé en 5.5 b, c'est-à-dire encore lorsque les états hauts et bas des impulsions ont la même durée, la valeur moyenne de la tension est égale à la moitié de l'amplitude totale des impulsions. Les ampoules ont alors une luminosité moyenne. Enfin, lorsque le rapport cyclique des impulsions est proche de 100 % comme schématisé en 5.5c, la valeur moyenne de la tension est proche de la tension d'alimentation totale et la luminosité des ampoules est maximale.

En première approximation on peut estimer que la tension moyenne est donnée par la relation :

$$V_{MOY} = V_A \times R_{CYC}$$

où V_A représente l'amplitude maximum des impulsions (5 volts le plus souvent dans notre cas), R_{CYC} le rapport cyclique exprimé en pour cent et V_{MOY} la tension moyenne ainsi obtenue.

Comparée à la solution faisant appel au rhéostat, la méthode PWM s'avère beaucoup plus performante car, pour produire des impulsions, on fait travailler un transistor en commutation et sa dissipation de puissance est alors quasi nulle. En effet, lorsque le transistor est bloqué, ce qui correspond aux états hauts des impulsions, il n'est traversé par aucun courant et ne dissipe donc aucune puissance. Lorsque ce même transistor est saturé, ce qui correspond aux états bas des impulsions, il est traversé par un fort courant (celui d'alimentation des ampoules) mais

© Dunod – La photocopie non autorisée est un délit.

la tension à ses bornes est très faible car c'est sa tension de saturation qui est en général inférieure au volt, et il ne dissipe donc qu'une puissance très faible.

Afin que vous puissiez constater de visu l'efficacité de ce procédé, nous vous proposons de réaliser le montage de la figure 5.6, soit au moyen d'un shield prototype soit, sans aucun câblage, grâce au Danger Shield présenté au chapitre 3.

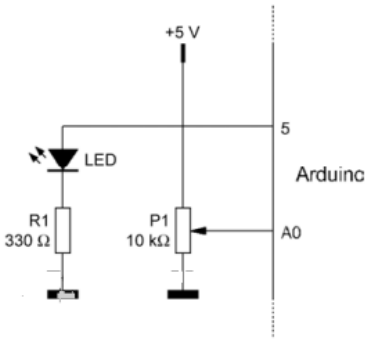

Figure 5.6 – Schéma de mise en pratique de la gradation de tension PWM.

Il permet de régler la luminosité d'une LED, simulant ici une charge plus puissante dont nous verrons dans un instant comment elle peut être commandée, en fonction de la position du curseur d'un potentiomètre.

Le programme nécessaire pour parvenir à un tel résultat est d'une grande simplicité comme vous pouvez le constater ci-dessous.

Listing 5.7 – Exemple de gradation de tension PWM.

```
#define potentiometre 0
#define LED 5

void setup()
{
  pinMode(LED, OUTPUT);
}

void loop()
{
  int luminosite = analogRead(potentiometre);
    // Lecture du potentiomètre
  luminosite = map(luminosite, 0, 1023, 0, 255);
    // Conversion dans la plage 0 - 255
  analogWrite(LED, luminosite);
    // Définition du signal PWM
}
```

On constate à nouveau dans ce programme l'intérêt de la fonction `map` qui permet ici de ramener la plage de sortie 0-1023 du convertisseur analogique/digital à la plage 0-255 qui est la seule admise par la fonction `analogWrite`.

Si la gradation de puissance d'une simple LED ne vous suffit pas, mais que vous souhaitez au contraire commander des charges plus puissantes, il suffit de faire appel à un amplificateur à transistor tels ceux que nous avons décrits au chapitre 4 lors de la présentation des sorties numériques. En effet, les signaux PWM étant des signaux logiques, les schémas que nous y avons présentés restent parfaitement utilisables ici. Ainsi, si vous voulez assurer la gradation de puissance d'une charge pouvant consommer jusqu'à 10 ampères, vous pourrez reprendre sans hésiter le schéma de la figure 4.12.

Le ou les signaux PWM définis au moyen de la fonction `analogWrite` sont générés de façon permanente suite à l'appel de cette fonction et ne sont interrompus que par un nouvel appel de cette fonction sur la même sortie, ou par la lecture ou l'écriture de données numériques sur la même sortie.

> **Attention !**
> Toutes les sorties numériques de l'Arduino ne peuvent pas générer de signaux PWM. Seules les sorties 3, 5, 6, 9, 10 et 11 sont dans ce cas sur le Uno tandis que l'on dispose de 2 à 13 sur le Mega 2560.

5.2.2 Tension continue à partir d'un signal PWM

Dans le cas de la gradation de puissance que nous venons de présenter, c'est l'organe commandé qui assure tout seul la conversion du signal PWM en un équivalent de la tension continue qu'il faudrait lui appliquer pour obtenir le même résultat. Pour parler à la façon des électroniciens, il réalise un filtrage passe-bas afin de disposer de la valeur moyenne du signal PWM.

Dans certaines applications, on peut ne pas vouloir faire de la gradation de puissance mais seulement disposer d'une tension continue variable à destination de tel ou tel équipement. Cela reste possible à partir d'une sortie PWM mais il faut alors réaliser nous-même ce filtrage passe-bas.

Une simple cellule RC, telle celle représentée figure 5.7a, suffit pour cet usage étant entendu que les valeurs de R et de C doivent être calculées en fonction de la charge connectée aux bornes de C et de la fréquence du signal PWM qui, dans le cas de l'Arduino, est de l'ordre de 490 Hz rappelons-le.

Afin d'améliorer la qualité de ce filtrage il est d'ailleurs vivement conseillé de faire appel à un amplificateur opérationnel monté en suiveur de tension comme indiqué figure 5.7b, ce qui permet de rendre les caractéristiques de la cellule RC indépendantes de la charge.

Cependant, et même en recourant à l'amplificateur suiveur, une telle conversion ne peut être qualifiée de précise. Si donc votre application nécessite une conversion digitale/analogique rigoureuse, le recours à un convertisseur externe s'avère nécessaire. Vous choisirez donc dans les nombreuses références proposées par les

© Dunod – La photocopie non autorisée est un délit.

fabricants spécialisés un modèle qui dispose d'une interface par bus I2C ou SPI par exemple, et le problème de conversion sera alors ramené à un simple dialogue sur l'un de ces deux bus qui sont présentés au chapitre suivant.

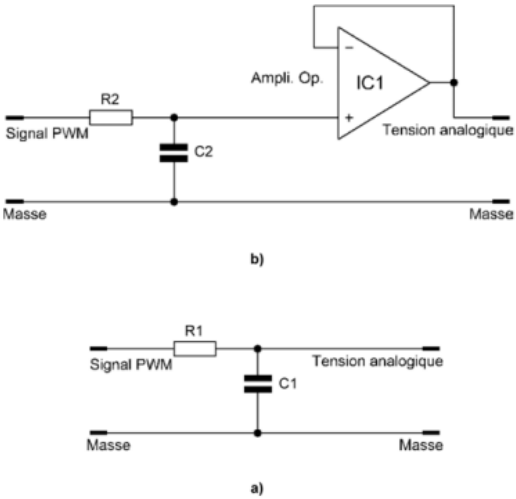

Figure 5.7 – Conversion d'un signal PWM en tension analogique.

5.3 La commande des moteurs

Même s'il est possible de commander des moteurs avec des relais, eux-mêmes pilotés par des sorties numériques comme nous l'avons vu au chapitre 4, une telle commande reste rudimentaire et se borne bien souvent à de simples fonctions marche/arrêt et inversion de sens. La diversité des moteurs que l'on rencontre aujourd'hui en électronique et en robotique permet de faire bien mieux en utilisant, selon le cas, des moteurs classiques à courant continu, des servomoteurs initialement conçus pour la radiocommande ou bien encore des moteurs pas à pas. Ce sont ces différentes familles de moteurs que nous vous proposons maintenant d'apprendre à faire tourner avec votre Arduino.

5.3.1 Les servos de radiocommande

Les servos de radiocommande nécessitent un mode d'interfaçage un peu particulier, pas tellement au plan électrique mais plutôt au plan logiciel. Voici tout à la fois le problème à résoudre et sa solution !

Un servo de radiocommande se présente sous un aspect physique similaire à ce que vous pouvez découvrir figure 5.8. Il est généralement fourni avec de nombreux accessoires pouvant être montés sur son arbre de sortie (disques perforés, bras de diverses tailles et formes, etc.) facilitant ainsi sa liaison avec les organes mécaniques environnants.

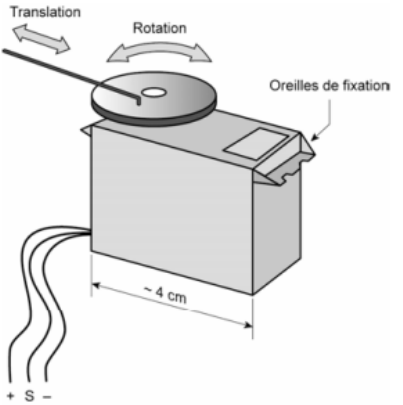

Figure 5.8 – Allure physique d'un servo de radiocommande.

Côté électrique, un tel servo ne dispose que de trois fils codés par des couleurs. Les fils rouge et noir sont destinés à son alimentation qui peut être comprise entre 4,8 et 6 volts.

Attention !
Compte tenu de la puissance des moteurs qui équipent les servos, un courant de plusieurs centaines de milli-ampère (mA) doit pouvoir être fourni par cette alimentation lorsque le servo tourne.

Le troisième fil, de couleur jaune ou blanche (ou de toute autre couleur que rouge et noir en fait), sert à transmettre les ordres au servo sous forme d'impulsions codées en largeur. La figure 5.9 montre le principe de codage de ces impulsions ; principe au demeurant fort simple.

Notez tout d'abord que ces dernières doivent se répéter à un rythme tel que l'on ne doit pas avoir plus de 10 à 20 ms d'écart entre deux impulsions successives.

– une impulsion de 1,5 ms de large place le servo dans sa position dite centrée ou de repos ;
– une impulsion de 1 ms de large fait tourner le servo dans sa position maximum en sens inverse des aiguilles d'une montre, ce qui représente généralement un angle de 90° par rapport à sa position de repos ;

© Dunod – La photocopie non autorisée est un délit.

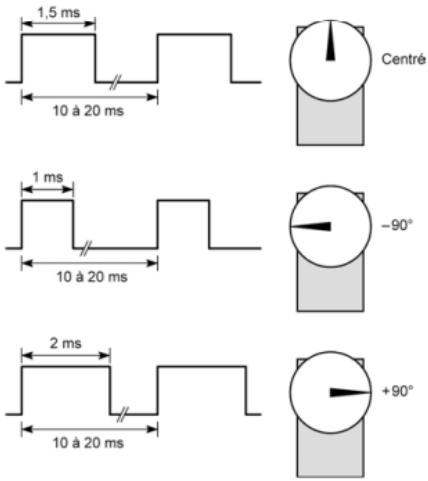

Figure 5.9 – Chronogramme de commande d'un servo de radiocommande.

– une impulsion de 2 ms de large fait tourner le servo dans sa position maximum dans le sens des aiguilles d'une montre ce qui représente, là aussi, un angle de 90° par rapport à sa position de repos ;
– toute autre largeur d'impulsion, comprise entre 1 et 2 ms, permet d'obtenir les innombrables positions intermédiaires.

La connexion d'un servo à l'Arduino est évidemment fort simple comme le montre la figure 5.10. La seule précaution à prendre se situe au niveau de l'alimentation du servo dont seule la masse doit être commune avec celle de l'Arduino. Par contre, la tension positive ne doit pas être prélevée sur le + 5 volts destiné à l'Arduino car, d'une part la consommation du servo est trop importante pour le régulateur intégré se trouvant sur la carte de l'Arduino, d'autre part chacun de ses mouvements provoque des appels de courant susceptibles de perturber l'Arduino en causant une baisse anormale de sa tension d'alimentation.

Ceci étant précisé, la commande d'un servo avec un Arduino est fort simple d'autant qu'une bibliothèque, installée d'origine avec l'environnement de développement, est disponible à cet effet. En effet, aucune des instructions de base du langage C de l'Arduino ne permet de générer facilement les impulsions nécessaires, et ce d'autant que ces dernières doivent se répéter à une cadence de 10 à 20 ms si l'on veut que le servo reste à la position qui lui a été définie.

La bibliothèque fournie est presque trop performante. Elle permet en effet de commander jusqu'à 12 servos avec le Uno et 48 servos avec le Mega 2560.

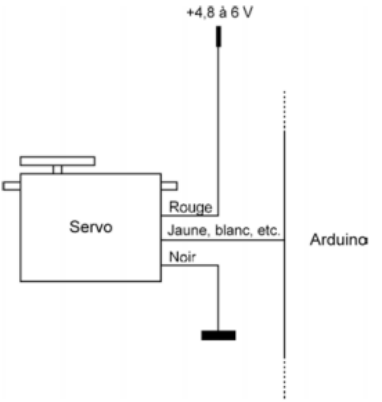

Figure 5.10 – Connexion d'un servo à un Arduino.

Attention !
Sur les Arduino autres que le Mega 2560, l'utilisation de cette bibliothèque invalide la fonction `analogWrite` sur les sorties 9 et 10 et ce, qu'un servo y soit connecté ou pas. Sur le Mega 2560, jusqu'à 12 servos peuvent être utilisés sans que cela interfère avec les fonctions PWM tandis que l'utilisation de servos sur les sorties 12 à 23 invalide la génération de signaux PWM sur les sorties 11 et 12.

Voici brièvement quelles sont les fonctions supportées par cette bibliothèque dont le nom est tout naturellement `Servo.h` ; fonctions qu'il n'est possible d'appeler qu'après avoir créé une variable (un objet pour parler correctement) de type `Servo` en écrivant par exemple :

```
Servo monServo ;
```

■ attach()

Cette fonction permet de définir, ou d'attacher, la connexion d'un servo à une sortie de l'Arduino. Elle permet également, de manière optionnelle, de définir les largeurs minimum et maximum des impulsions admises par le servo. Elle s'utilise donc sous l'une des deux formes suivantes :

```
monServo.attach(sortie) ;
monServo.attach(sortie, minimum, maximum) ;
```

`monServo` est le nom préalablement donné à l'objet `Servo` comme indiqué ci-dessus tandis que `sortie` est le numéro de la sortie utilisée. `minimum`, dont la valeur par défaut est de 544, et `maximum`, dont la valeur par défaut est de 2 400, sont respectivement les largeurs minimum et maximum des impulsions envoyées au servo, exprimées en microseconde (µs).

© Dunod – La photocopie non autorisée est un délit.

■ detach()

C'est évidemment le contraire de la fonction précédente qui permet de détacher un servo de la sortie à laquelle il a été préalablement attaché. L'utilisation en est fort simple :

```
monServo.detach(sortie) ;
```

■ write()

C'est la fonction principale de commande du servo puisqu'elle permet de faire générer, sur la sortie préalablement attachée à un servo, le train d'impulsions permettant de commander celui-ci. Afin de faciliter la gestion de la position du servo, le paramètre utilisé n'est pas la largeur de l'impulsion générée mais la position angulaire du servo qui peut varier de 0 à 180°, sachant que 90° est la position de repos ou position médiane. La syntaxe est la suivante :

```
monServo.attach(sortie) ;
monServo.write(position) ;
```

où `position` est une variable de type `int` comprise entre 0 et 180 et correspond à la position angulaire du servo.

> **Attention !**
> Sur les servos modifiés pour tourner de façon continue, tels ceux que l'on utilise par exemple pour propulser de petits robots, `position` ne fixe plus la position du servo, puisque cela n'a plus aucun sens, mais la vitesse de rotation de celui-ci. 0 correspond alors à la vitesse maximum dans uns sens, 180 à la vitesse maximum dans l'autre sens et 90 à l'arrêt.

■ writeMicroseconds()

Cette fonction agit comme `write` mais, au lieu de définir la position angulaire du servo, elle permet de définir la largeur de l'impulsion générée. Cette valeur est exprimée en micoseconde (µs) et varie usuellement de 1 000 à 2 000 comme nous l'avons vu figure 5.9 ; la position centrale ou de repos étant obtenue pour 1 500. Notez cependant que certains servos peuvent aller au-delà de ces positions et que la variable utilisée par cette fonction n'est donc pas contrainte à cette seule plage de variation. La syntaxe de cette fonction est tout simplement :

```
monServo.attach(sortie) ;
monServo.writeMicroseconds(largeur) ;
```

où `largeur` est une variable de type `int` comprise habituellement entre 1 000 et 2 000 et correspond à la largeur de l'impulsion générée, exprimée en microseconde (µs).

■ read()

D'usage moins courant, cette fonction permet de connaître la position d'un servo en renvoyant la valeur utilisée lors du dernier appel à la fonction `write` précédent.

Elle s'utilise sous la forme :

```
position = monServo.read() ;
```

`position` est compris entre 0 et 180 et est égale à la dernière variable de positionnement du servo qui lui a été envoyée via la fonction `write`.

L'utilisation de ces fonctions s'avère donc être d'une grande simplicité comme le montre l'exemple ci-dessous, que vous pourrez mettre en œuvre par exemple au moyen d'un shield prototype tels ceux vus au chapitre 3.

Il a pour but de faire tourner un servo, connecté sur la sortie 7 d'un Arduino Uno, par pas de 1°, de sa position 0° à sa position 180° et retour.

Listing 5.8 – Exemple de commande d'un servo au moyen de la bibliothèque prévue à cet effet.

```
#include <Servo.h>

Servo monServo;    // Création de l'objet monServo
int position = 0 ; // Initialisation de position

void setup()
{
  monServo.attach(7);       // Servo connecté à la sortie 7
  monServo.write(position); // Servo positionné à 0°
}

void loop()
{
  for(position = 0; position < 180; position ++)
    // Evolution de 0 à 180°
  {
    monServo.write(position);
    delay(200);// Délai de stabilisation du servo
  }
  for(position = 180; position >= 1; position --)
    // Evolution de 180° à 0
  {
    monServo.write(position);
    delay(200); // Délai de stabilisation du servo
  }
}
```

Ce listing se passe de tout commentaire puisque l'on y retrouve tout simplement les fonctions que nous venons de décrire, utilisées de manière strictement conforme à la syntaxe que nous avons présentée.

Et puisqu'un servo est fréquemment utilisé pour recopier une position, nous vous proposons un autre exemple mettant en œuvre tout à la fois, la lecture d'une tension analogique issue d'un potentiomètre, et la commande d'un servo. Le schéma utilisé est visible figure 5.11 et peut être réalisé très facilement à partir d'un Danger Shield par exemple.

© Dunod – La photocopie non autorisée est un délit.

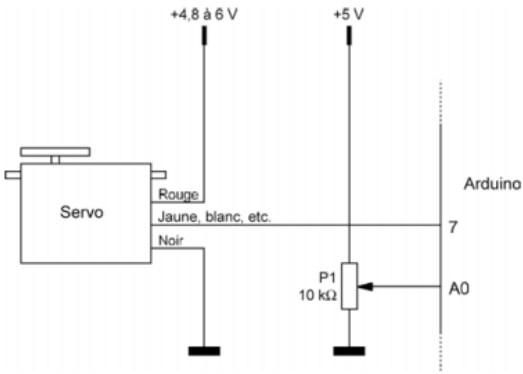

Figure 5.11 – Commande de la position d'un servo au moyen d'un potentiomètre.

Le programme nécessaire pour piloter un tel montage vous est proposé sur le listing suivant.

Listing 5.9 – Commande de la position d'un servo au moyen d'un potentiomètre.

```
#include <Servo.h>
#define potentiometre 0

Servo monServo;     // Création de l'objet monServo
int position = 0 ;  // Initialisation de position
int tension ;

void setup()
{
  monServo.attach(7);        // Servo connecté à la sortie 7
  monServo.write(position);  // Servo positionné à 0°
}

void loop()
{
  tension = analogRead(potentiometre);
  // Lecture de la position du potentiomètre
  position = map(tension, 0, 1023, 0, 180) ;
  // Restriction à la plage 0 - 180
  monServo.write(position);
  delay(20) ;
}
```

Après avoir défini la sortie à laquelle est connecté le servo et avoir initialisé la position de celui-ci à 0°, ce qui n'est bien sûr nullement indispensable, la tension délivrée par le potentiomètre est convertie en numérique et récupérée dans la variable `tension`. La plage de variation de cette dernière est ensuite ramenée à 0

–180, ce qui correspond à la plage autorisée pour la variable `position` utilisée par la fonction `write` de la bibliothèque `Servo`. On apprécie ici encore l'utilité de la fonction `map` qui permet de réaliser très facilement une telle opération.

5.3.2 Les moteurs pas à pas

Un autre mode d'action mécanique de mise en œuvre très courante consiste à faire appel à des moteurs pas à pas. Contrairement aux moteurs à courant continu ou alternatif classiques qui tournent tant qu'ils sont alimentés, les moteurs pas à pas ne tournent que lorsqu'ils reçoivent des impulsions sur leurs bobines ; impulsions qui doivent être présentées dans un ordre bien précis pour faire tourner le moteur dans un sens ou dans l'autre.

Du fait de cette commande impulsionnelle, les moteurs ne tournent pas de manière continue mais avancent en fait à chaque fois d'un pas élémentaire, qui varie selon le type de moteur entre 1,8° et 7,5° le plus souvent. Cette avance par pas permet au programme qui commande le moteur de connaître très précisément sa position, sous réserve bien sûr que la charge maximum du moteur ne soit pas dépassée, ce qui le conduirait alors à ne pas avancer, même en ayant reçu les impulsions adéquates.

En outre, si le moteur est alimenté mais ne reçoit plus d'impulsions, il reste verrouillé sur la position qu'il a atteinte. On dispose ainsi d'une sorte de frein électrique, toujours sous réserve de ne pas dépasser les possibilités de charge du moteur bien sûr.

Les moteurs pas à pas les plus simples sont les moteurs dits unipolaires à quatre bobines. Pour les commander, il suffit en effet d'appliquer ou non une tension à leurs bobines. Dans l'autre variété de moteurs, appelés moteurs bipolaires, il faut inverser régulièrement la polarité de la tension appliquée aux bobines ce qui complique le schéma à mettre en œuvre.

Tableau 5.1 – Ordre d'alimentation des bobines d'un moteur pas à pas unipolaire.

	Numéro de pas				
	1	**2**	**3**	**4**	**5**
Bobine 1	1	1	0	0	1
Bobine 2	0	0	1	1	0
Bobine 3	1	0	0	1	1
Bobine 4	0	1	1	0	0

Le tableau 5.1 présente l'ordre dans lequel il faut alimenter les bobines d'un moteur unipolaire pour le faire tourner. Si l'on parcourt les séquences de ce tableau de 1 vers 5 on fait tourner le moteur dans le sens des aiguilles d'une montre, alors qu'un parcours inverse change son sens de rotation. En outre, chaque pas de ce tableau

© Dunod – La photocopie non autorisée est un délit.

correspond à un pas mécanique du moteur compris entre 1,8 ° et 7,5 ° selon les modèles.

Il existe sur le marché divers circuits spécialisés de commande de moteurs pas à pas mais, dans le cas de l'Arduino, ils n'induisent ni économie de programme ni économie de composants externes. Nous vous proposons donc une solution beaucoup plus simple utilisant un banal ULN 2803 déjà utilisé au chapitre 4 lors de la commande de plusieurs relais. Le coût de ce circuit est en effet nettement inférieur à celui des circuits spécialisés pour la commande des moteurs pas à pas et le logiciel nécessaire pour le faire fonctionner n'est pas plus complexe.

La méthode de commande la plus simple qui vient à l'esprit en présence d'un moteur pas à pas unipolaire à quatre bobines est celle schématisée figure 5.12.

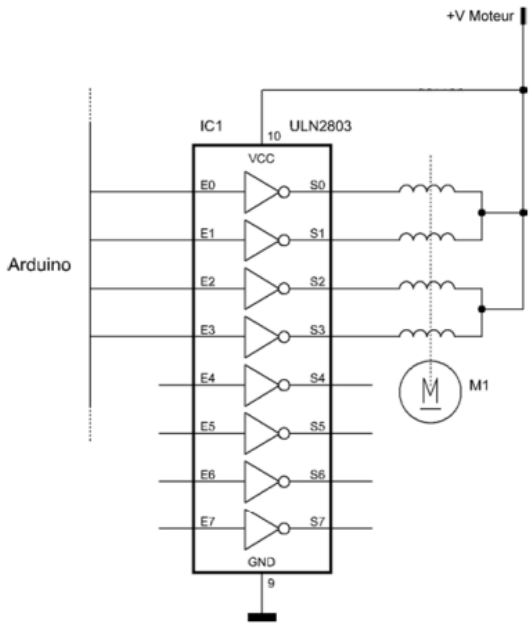

Figure 5.12 – Commande d'un moteur pas à pas unipolaire à quatre bobines au moyen de quatre sorties de l'Arduino.

Comme on le voit sur cet exemple, chaque bobine est commandée par une sortie de l'Arduino ; l'ULN 2803 faisant office d'amplificateur de courant permettant ainsi de commander des moteurs consommant jusqu'à 500 mA par bobine et dont la tension de fonctionnement peut atteindre 48 volts.

Le logiciel de commande se borne à générer sur les sorties les niveaux indiqués par le tableau 5.1, en le parcourant de gauche à droite ou de droite à gauche selon que l'on souhaite faire tourner le moteur dans un sens ou dans l'autre.

Ce schéma fonctionne mais présente l'inconvénient de consommer quatre sorties de l'Arduino alors que l'examen détaillé du tableau 5.1 permet de constater que l'état des bobines 1 et 4 d'une part et 2 et 3 d'autre part est toujours contraire. Ceci permet de commander le moteur pas à pas avec seulement deux sorties de l'Arduino comme le montre le schéma fort simple de la figure 5.13. Les amplificateurs libres contenus dans l'ULN 2803 servent en effet d'inverseurs pour commander les bobines 2 et 3 à partir de l'état des bobines 1 et 4.

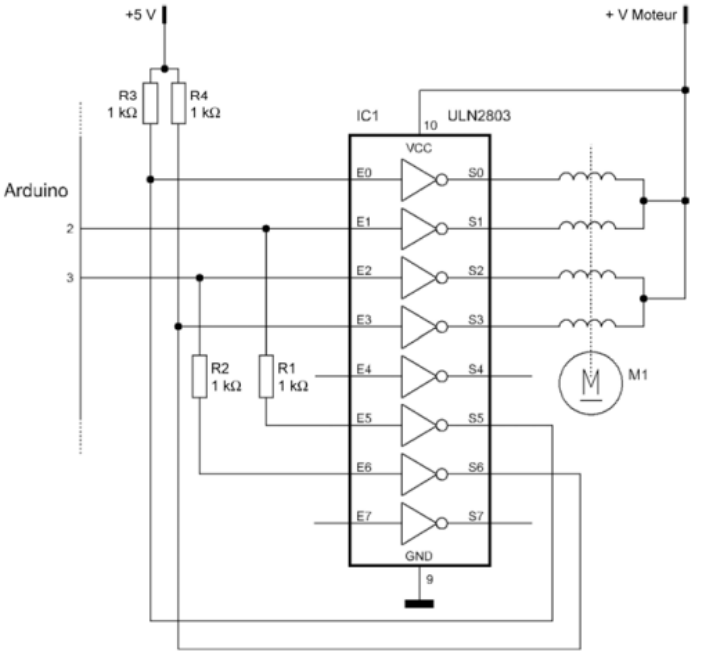

Figure 5.13 – Commande d'un moteur pas à pas unipolaire à quatre bobines avec seulement deux sorties de l'Arduino.

Ici aussi, il ne reste plus qu'à générer sur ces sorties les niveaux logiques obtenus en parcourant de gauche à droite ou de droite à gauche le tableau de la figure 5.1 selon que l'on souhaite faire tourner le moteur dans un sens ou dans l'autre.

© Dunod – La photocopie non autorisée est un délit.

Et pour que la génération des signaux correspondant soit la plus facile possible, une bibliothèque spécialisée est intégrée d'origine à l'installation de l'environnement de développement de l'Arduino.

Voici brièvement quelles sont les fonctions supportées par cette bibliothèque dont le nom est tout naturellement `Stepper.h`.

■ Stepper()

Cette fonction doit impérativement être appelée avant toute utilisation des autres fonctions de la bibliothèque car, outre le fait qu'elle définisse un certain nombre de paramètres comme indiqué ci-après, c'est également elle qui crée la variable (l'objet) de type `Stepper`. Elle doit donc être placée préalablement à la fonction `setup()` du programme.

Elle s'utilise sous deux formes différentes, selon que le moteur est connecté à quatre sorties de l'Arduino comme indiqué figure 5.12 ou à seulement deux sorties comme indiqué figure 5.13. Elle permet également d'indiquer au logiciel combien de pas sont nécessaires pour que le moteur fasse un tour complet. Cette donnée étant rarement fournie dans les caractéristiques d'un moteur pas à pas, dont on indique plus volontiers la taille des pas ; elle s'obtient tout simplement en divisant 360° par la taille individuelle d'un pas. Ainsi, un classique moteur de 1,8° par pas nécessite-t-il 200 pas pour faire un tour complet. Les deux syntaxes possibles sont les suivantes :

```
Stepper PasPas (nombre de pas, sortie1, sortie2, sortie3, sortie4) ;
Stepper PasPas (nombre de pas, sortie1, sortie2) ;
```

Ceci a pour effet de créer une variable de type Stepper, baptisée ici `PasPas` (c'est très original !) dont les paramètres ont la signification suivante :

– `Nombre de pas` est une variable de type `int` spécifiant le nombre de pas nécessaires pour faire un tour complet comme indiqué ci-dessus.
– `sortie1, sortie2` et `sortie3, sortie4` le cas échéant sont les numéros des sorties de l'Arduino utilisées pour commander le moteur selon les schémas des figures 5.12 ou 5.13.

■ setSpeed()

Comme son nom l'indique, cette fonction permet de définir la vitesse de rotation du moteur c'est-à-dire aussi la fréquence de génération des signaux rectangulaires appliqués à ses bobines. Elle s'utilise sous la forme :

```
Stepper PasPas (nombre de pas, sortie1, sortie2) ;
PasPas.setSpeed (vitesse) ;
```

`vitesse` est une variable positive de type `long` spécifiant la vitesse de rotation du moteur exprimée en tours par minute.

■ steps()

C'est évidemment la fonction la plus utile de cette bibliothèque puisque c'est celle qui permet de faire tourner le moteur du nombre de pas spécifié. Elle s'utilise comme suit :

```
Stepper PasPas (nombre de pas, sortie1, sortie2) ;
PasPas.steps (pas) ;
```

`pas` est une variable de type `int`, spécifiant le nombre de pas à générer. Cette variable peut être positive ou négative selon le sens dans lequel on souhaite faire tourner le moteur.

Comme vous pouvez le constater, ces fonctions sont très simples, ce que confirme l'exemple de programme ci-dessous qui réalise la même fonction que celle que nous avons accomplie avec le listing 5.9 pour la commande d'un servo.

Ce programme lit donc la position d'un potentiomètre connecté à l'entrée analogique A0 de l'Arduino et recopie sa position sur un moteur pas à pas unipolaire à quatre bobines, câblé comme indiqué figure 5.13 sur les sorties 2 et 3 d'un Arduino Uno.

Listing 5.10 – Commande de la position d'un moteur pas à pas au moyen d'un potentiomètre.

```
#include <Stepper.h>
#define potentiometre 0
#define nbpas 200    // Nombre de pas pour faire un tour
#define vitesse 20   // Vitesse du moteur

Stepper PasPas(nbpas ,2 ,3) ; // Création de l'objet PasPas
                              // Définition de son mode de connexion

int position = 0 ;            // Initialisation de position
int positionactuelle = 0;     // Initialisation de positionactuelle
int tension ;

void setup()
{
  PasPas.setSpeed(vitesse);  // Définition de la vitesse du moteur
}

void loop()
{
  tension = analogRead(potentiometre);
  // Lecture de la position du potentiomètre
  position = map(tension, 0, 1023, 0, 200) ;
  // Restriction à la plage 0 - 200
  PasPas.step(position - positionactuelle);
  positionactuelle = position ;
  // La position devient la positionactuelle
  delay(20) ;
}
```

Après avoir défini le mode de connexion du moteur lors de la création de l'objet `PasPas`, sa vitesse de rotation est fixée dans la fonction `setup`. Il ne reste plus alors qu'à lire la position du potentiomètre puis à restreindre la valeur ainsi obtenue à la plage correspondant au nombre de pas maximum nécessaire pour faire un tour, au moyen de la fonction `map` désormais bien connue. Le résultat est envoyé au moteur sous la forme de la différence entre la position demandée et la position actuelle

© Dunod – La photocopie non autorisée est un délit.

atteinte par le moteur étant entendu que, grâce à la ligne suivante, la position ainsi atteinte devient la nouvelle position actuelle.

5.3.3 Les moteurs à courant continu

Lorsque, ni les servos de radiocommande, ni les moteurs pas à pas ne conviennent et qu'il faut donc faire appel à un moteur à courant continu, sa commande avec un Arduino reste possible mais peut nécessiter, selon le cas, un circuit spécialisé externe.

Si le moteur ne doit tourner que dans un seul sens, et que l'on ne souhaite donc pouvoir doser à partir de l'Arduino que sa vitesse de rotation, l'utilisation de la fonction `analogWrite` telle qu'elle a été décrite au § 5.2.1 ci-avant convient parfaitement.

Il ne reste qu'à mettre en place sur la sortie de l'Arduino qui commande le moteur un amplificateur de courant tels ceux que nous avons déjà vus à plusieurs reprises et dont les deux principaux schémas d'utilisation sont rappelés figure 5.14.

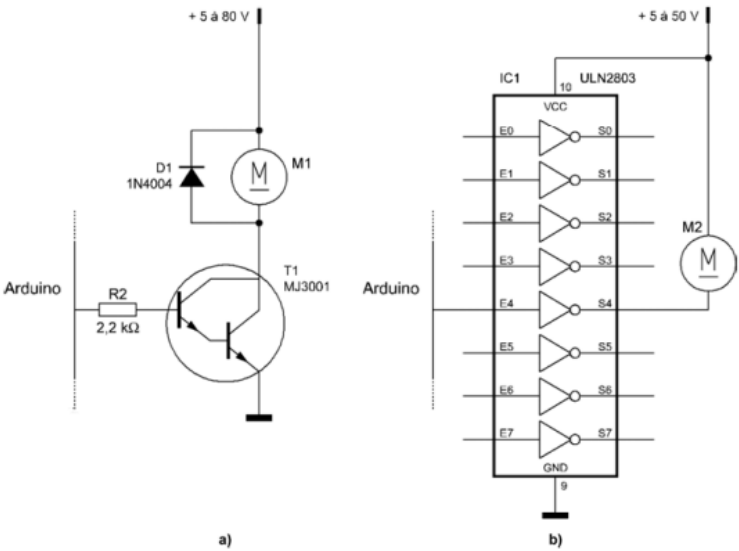

Figure 5.14 – Commande d'un moteur à courant continu avec un seul sens de rotation.

La figure 5.14b fait appel à un ULN2803, désormais bien connu, dont nous rappelons qu'il peut fournir au maximum 500 mA sous une tension maximum absolue de 50 volts. La figure 5.14a quant à elle utilise un transistor darlington de

puissance très répandu qui peut commander jusqu'à 10 ampères sous une tension maximum de 80 volts.

Par contre, si le sens de rotation du moteur doit pouvoir être modifié par logiciel, ces schémas ne sont plus utilisables tels quels. Il est bien sûr possible de leur adjoindre des relais pour assurer l'inversion de sens mais, compte tenu de la complexité, de l'encombrement et du surcoût que cela engendre, il est alors préférable de faire appel à un circuit spécialisé au premier rang desquels se classe le L298 de ST Microelectronics.

Ce circuit, dont vous trouverez la fiche technique complète dans les compléments en ligne de cet ouvrage, est un double pilote de moteurs à courant continu capable de commander tout moteur consommant jusqu'à un maximum de 2 ampères sous une tension pouvant atteindre 48 volts.

Il se commande par des signaux logiques et peut donc être relié directement aux sorties de l'Arduino et ne nécessite, comme composants externes, que quatre diodes de protection par moteur. La figure 5.15 montre son utilisation typique dans le cas où l'on souhaite pouvoir inverser le sens de rotation d'un moteur et où l'on veut pouvoir doser sa vitesse au moyen de signaux PWM.

Comme le montre cette figure, on dispose de deux signaux de commande. Celui appliqué sur In1 permet de déterminer le sens de rotation du moteur selon qu'il est à un niveau logique haut ou bas. Celui appliqué sur EnA commande quant à lui la vitesse de rotation du moteur au moyen d'un signal PWM.

Notez dès à présent que le L298 est double et comporte donc, dans un seul boîtier, deux fois le circuit de la figure 5.15. Il peut donc commander simultanément et indépendamment l'un de l'autre deux moteurs à courant continu. C'est donc un circuit idéal pour exploiter, par exemple, une base robotique à deux moteurs tel celles que l'on trouve aujourd'hui couramment sur le marché.

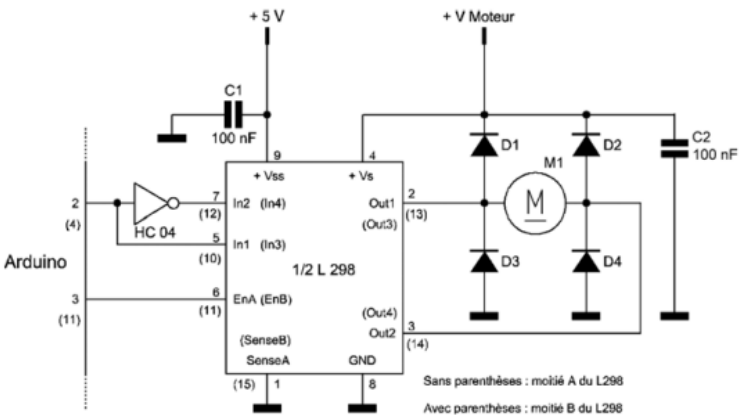

Figure 5.15 – Commande bidirectionnelle d'un moteur à courant continu au moyen d'un L298.

© Dunod – La photocopie non autorisée est un délit.

Si vous souhaitez utiliser un L298, vous pouvez bien sûr le câbler sur un shield prototype mais il existe une solution encore plus simple qui consiste à faire appel à l'Ardumoto Motor Shield, commercialisé par Sparkfun (*www.sparkfun.com*) et distribué en France par Lextronic (*www.lextronic.fr*).

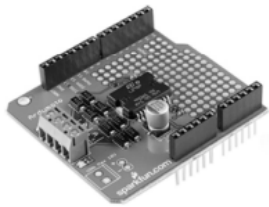

Figure 5.16 – L'Ardumoto Motor Driver Shield de Sparkfun (doc. Sparkfun).

Ce shield, dont l'aspect est visible figure 5.16 et le schéma figure 5.17, supporte en effet un L298 associé à ses diodes de protection et aux deux inverseurs nécessaires pour réaliser un montage similaire à ce que nous avons présenté figure 5.15.

Comme la consommation des moteurs qu'il est susceptible de commander peut être importante, un double bornier à vis est prévu pour raccorder ces derniers et, même s'il n'est pas implanté sur la photo de la figure 5.16, un deuxième bornier est prévu pour recevoir l'alimentation du L298 et donc des moteurs qu'il commande. Il est en effet hors de question de prélever celle-ci à partir des régulateurs de tension de l'Arduino.

Aucune bibliothèque spécialisée n'est nécessaire pour commander le L298 puisque le choix du sens de rotation des moteurs a lieu par simple mise au niveau haut ou bas des entrées DIRA et DIRB du L298 tandis que la vitesse de rotation est déterminée par le rapport cyclique des signaux appliqués sur PWMA et/ou PWMB. Tout ce que nous avons écrit au § 5.2.1 reste donc valable à ce sujet comme vous pouvez le constater avec l'exemple très simple présenté ci-après. Il est directement extrait du site de Sparkfun et fait tourner les moteurs connectés à l'Ardumoto Motor Driver Shield (ou à tout shield réalisant le même schéma) à vitesse réduite dans un sens, à pleine vitesse dans le même sens, puis en sens inverse puis à nouveau à vitesse réduite en sens inverse.

Listing 5.11 – Exemple de commande de moteurs à courant continu avec un L298.

```
int pwm_a = 3;  // Entrée PWMA du L298 sur 3
int pwm_b = 11; // Entrée PWMB du L298 sur 11
int dir_a = 2;  // Entrée DIRA du L298 sur 2
int dir_b = 4;  // Entrée DIRB du L298 sur 4

void setup()
{
  pinMode(pwm_a, OUTPUT);
  pinMode(pwm_b, OUTPUT);
```

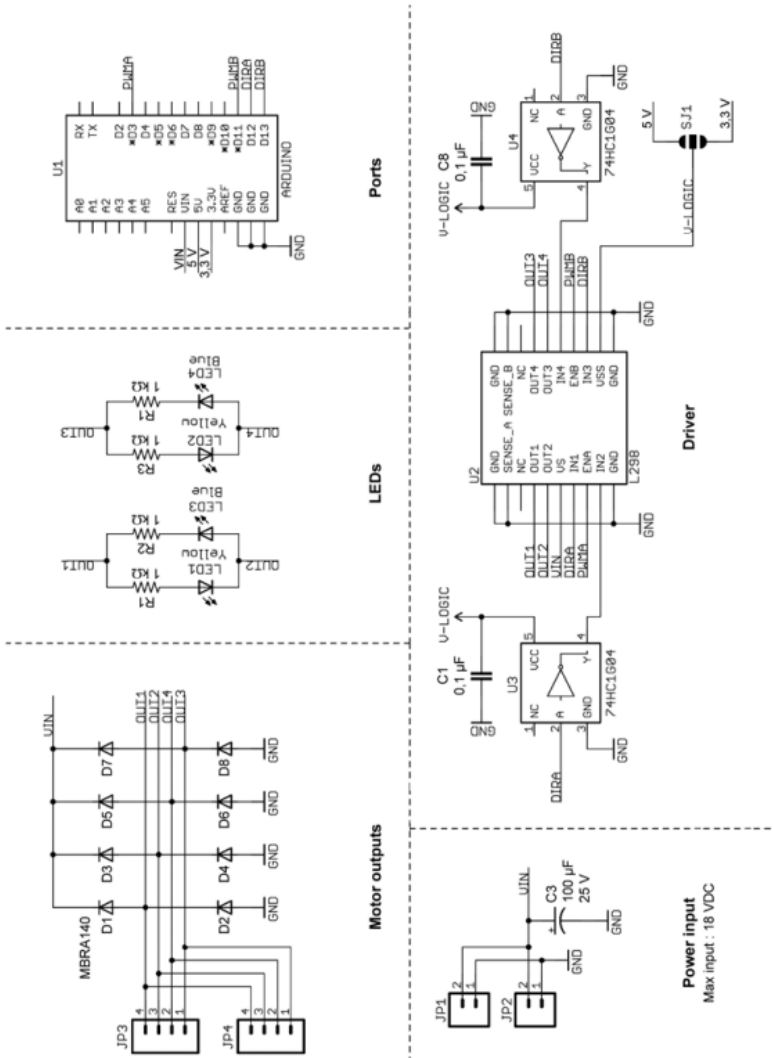

Figure 5.17 – Schéma de l'Ardumoto Motor Driver Shield de Sparkfun (doc. Sparkfun).

© Dunod – La photocopie non autorisée est un délit.

```
  pinMode(dir_a, OUTPUT);
  pinMode(dir_b, OUTPUT);

  analogWrite(pwm_a, 127); // Moteur A à 50 % de sa vitesse maxi
  analogWrite(pwm_b, 127); // Moteur B à 50 % de sa vitesse maxi
}

void loop()
{
  digitalWrite(dir_a, LOW); // Moteur A dans un sens
  digitalWrite(dir_b, LOW); // Moteur B dans le même sens

  delay(1000);

  analogWrite(pwm_a, 255);  // Moteur A à vitesse maxi
  analogWrite(pwm_b, 255);  // Moteur B à vitesse maxi

  delay(1000);

  digitalWrite(dir_a, HIGH); // Changement de sens Moteur A
  digitalWrite(dir_b, HIGH); // Changement de sens Moteur B

  delay(1000);

  analogWrite(pwm_a, 100);  // Retour à mi-vitesse Moteur A
  analogWrite(pwm_b, 100);  // Retour à mi-vitesse Moteur B

  delay(1000);
}
```

Nous vous laissons le soin d'examiner ce listing à la lumière de ce que nous vous avons expliqué au § 5.2.1 car ce n'en est que la mise en pratique.

6 • LES INTERFACES SÉRIES SYNCHRONES ET ASYNCHRONES, I2C, SPI, BUS « UN FIL »

Même si l'utilisation des ports parallèles vue au chapitre 4 permet d'interfacer les Arduino avec de nombreux éléments externes, elle est loin de résoudre tous les problèmes de communication que l'on peut rencontrer au sein d'une application. En effet, de nombreux circuits, capteurs ou équipements informatiques utilisent pour communiquer ce que l'on peut regrouper sous le vocable commun de liaisons séries ou de bus séries selon le cas. Fort heureusement, ces bus ou liaisons sont aujourd'hui parfaitement normalisés pour ce qui est des plus importants d'entre eux et il est donc relativement facile d'y connecter un Arduino.

Ce chapitre va donc s'intéresser à ces ressources et nous allons y étudier comment communiquer avec « le monde extérieur » via une liaison série asynchrone, un bus I2C, une interface SPI ou bien encore un bus « un fil ». Grâce aux exemples fournis dans ce chapitre, vous serez alors à même d'interfacer votre Arduino avec quasiment tous les équipements, capteurs ou circuits intégrés spécialisés actuellement commercialisés.

6.1 Les liaisons séries asynchrones

La liaison série asynchrone, très souvent appelée RS 232 par un raccourci qui confond volontairement les niveaux électriques utilisés et le mode de fonctionnement de cette dernière, permet de relier une application à tous les équipements informatiques qui en sont équipés.

6.1.1 Interface matérielle ou logicielle

L'exploitation d'une telle liaison a lieu en principe grâce à un UART ou USART, et tous les microcontrôleurs Atmel de la famille AVR ATmega qui équipent les Arduino en disposent. La fonction `Serial`, présentée au chapitre 2, exploite d'ailleurs cet UART pour gérer la liaison série asynchrone de l'Arduino.

Tout irait donc pour le mieux si l'interface série de l'Arduino était à disposition de votre application mais ce n'est hélas que partiellement le cas, au moins sur les Arduino Uno. En effet, c'est aussi cette interface qui, par le biais d'un convertisseur USB – série, assure la liaison avec le PC via justement l'interface USB.

© Dunod – La photocopie non autorisée est un délit.

Si donc vous avez besoin d'une interface série asynchrone réellement autonome et indépendante, ou bien encore si, dans une même application, vous avez besoin de plusieurs interfaces séries, il ne vous reste que deux solutions :

– la première est de faire appel à un Arduino Mega 2560 qui dispose, en plus de l'interface destinée au port USB, de trois autres interfaces séries totalement indépendantes ;
– la seconde, si vous tenez à tout prix à utiliser un Uno, est de faire appel à une interface série asynchrone par logiciel. Elle vous permettra alors de disposer d'une ou plusieurs autres interfaces de ce type sur n'importe quelles entrées/sorties numériques de l'Arduino, mais avec quelques restrictions.

Dans cette deuxième situation en effet, une bibliothèque spécialisée, fournie d'origine avec l'environnement de développement, permet de générer un pseudo UART par logiciel, c'est-à-dire en fait de transformer deux lignes quelconques de ports parallèles en entrée et sortie série asynchrone au moyen d'un programme adéquat. Les performances qu'il faut en attendre ne sont toutefois pas celles permises si l'on utilise un véritable UART matériel, et le programme nécessaire occupe une place non négligeable en mémoire.

Nous ne reviendrons pas ici sur les fonctions de `Serial`, qui gère donc l'UART matériel des Uno, ou de tous les UART matériels du Mega 2560, vous renvoyant pour cela au § 2.9 où nous avons décrit en détail ses possibilités. Rappelons cependant que, du fait de l'utilisation de l'UART matériel intégré dans le microcontrôleur, `Serial` est capable de fonctionner en tâche de fond c'est-à-dire par exemple de recevoir des caractères alors que le programme principal exécuté par l'Arduino fait tout autre chose. Ceci est rendu possible parce que l'UART matériel dispose d'un tampon (un buffer si vous préférez cette appellation) de 128 caractères.

Ceci étant précisé, nous allons maintenant nous intéresser à la bibliothèque série par logiciel afin de voir ce que l'on peut en attendre.

Avant la mise à disposition de la version 1.0 de l'environnement de développement, cette bibliothèque présentait un certain nombre de contraintes, largement présentées dans la précédente édition de cet ouvrage. Fort heureusement, la majorité d'entre elles a aujourd'hui disparu et la bibliothèque SoftwareSerial, puisque tel est son nom, peut être utilisée avec presque autant de souplesse que la version matérielle.

La limitation principale la concernant, si on ne s'intéresse qu'à son usage sur l'Arduino Uno, est que, dans le cas de l'ouverture de plusieurs ports séries virtuels au moyen de cette bibliothèque, il n'est pas possible de recevoir simultanément des données sur plusieurs d'entre eux.

Ceci étant précisé, voici quelles sont les fonctions principales à connaître pour pouvoir utiliser cette bibliothèque.

Attention !

Si vous « récupérez » sur Internet des programmes utilisant cette bibliothèque, il se peut que certains d'entre eux ne fassent pas appel à certaines des fonctions ci-dessous et utilisent des astuces de programmation plus ou moins complexes pour émuler ces dernières. Si tel est le cas, cela signifie tout simplement que ces programmes ont été écrits pour la version précédente de SoftwareSerial et n'ont pas été mis à jour.

■ SoftwareSerial()

Cette fonction doit impérativement être appelée avant toute utilisation des autres fonctions de la bibliothèque car, outre le fait qu'elle définisse les lignes utilisées pour l'entrée et la sortie série, c'est également elle qui crée la variable (l'objet) de type `SoftwareSerial`. Elle doit donc être placée préalablement à la fonction `setup()` du programme et s'utilise de la façon suivante :

```
SoftwareSerial MonPortSerie = SoftwareSerial(entréeRx, sortieTx) ;
```

où `MonPortSerie` est le nom donné à l'objet port série asynchrone par logiciel, tandis que `entréeRx` est le numéro de la ligne d'entrée numérique utilisée pour la réception de données et `sortieTx` le numéro de la ligne de sortie numérique utilisée pour l'émission de données.

■ begin()

Cette fonction doit être appelée au moins une fois, généralement dans la fonction `setup` du programme, afin de définir la vitesse utilisée sur la liaison série. La syntaxe est fort simple :

```
MonPortSerie.begin(vitesse) ;
```

`vitesse` peut prendre l'une des valeurs de vitesses normalisées suivantes : 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 31250, 38400, 57600 et 115200. `MonPortSerie` est évidemment le nom donné à la liaison série par logiciel lors de l'appel de `SoftwareSerial`.

■ available ()

Comme son nom le laisse supposer, cette fonction indique le nombre de caractères déjà reçus et donc disponibles pour la lecture sur le port série virtuel. Elle s'utilise sous la forme :

```
caracteresdispo = MonPortSerie.available() ;
```

`caracteresdispo`, qui est nécessairement de type `int`, contient alors le nombre de caractères disponibles et en attente d'être lus au niveau du port série virtuel.

■ overflow ()

Cette fonction indique un débordement éventuel au niveau du tampon de réception du port série virtuel. Ce dernier ne peut en effet contenir que 64 caractères et, si ceux-ci ne sont pas lus assez vite, ils sont écrasés par les caractères suivants. Elle s'utilise sous la forme :

```
debordement = MonPortSerie.overflow() ;
```

`debordement`, qui est nécessairement de type `boolean`, est égal à zéro tant qu'il n'y a pas de débordement. L'appel de cette fonction remet automatiquement à zéro l'indicateur de débordement.

© Dunod – La photocopie non autorisée est un délit.

- **read ()**

  Comme son nom le laisse supposer cette fonction lit le dernier caractère n'ayant pas encore été lu depuis le tampon de réception du port série virtuel. Si plus aucun caractère n'est disponible, elle renvoie la valeur –1. Elle s'utilise sous la forme :

  ```
  caractère = MonPortSerie.read() ;
  ```

  `caractère`, qui est nécessairement de type `int`, contient alors le caractère lu, ou –1 si aucun caractère n'est disponible.

- **listen ()**

  Cette fonction permet de définir quel port série virtuel doit se mettre à « écouter » son entrée afin d'y recevoir des caractères, étant entendu qu'il n'est pas possible de recevoir simultanément des données sur plusieurs ports séries virtuels comme nous l'avons indiqué précédemment. Elle s'utilise sous la forme :

  ```
  MonPortSerie.listen() ;
  ```

  où `MonPortSerie` est le nom du port série virtuel qui doit recevoir les données.

- **isListening ()**

  Cette fonction est en quelque sorte le complément de la précédente car elle permet de savoir quel port série virtuel est en cours d'écoute. Elle s'utilise sous la forme :

  ```
  ecoute = MonPortSerie.isListening() ;
  ```

  `ecoute`, qui est nécessairement de type `boolean`, est différent de zéro si `MonPortSerie` est à l'écoute.

- **print() et println()**

  Ces fonctions permettent de faire sortir sur la liaison série une ou plusieurs données et de préciser leur mode de codage afin de faciliter leur affichage sur un terminal, sur un afficheur alphanumérique à cristaux liquides ou bien encore sur tout dispositif acceptant en entrée du code ASCII. Elles fonctionnent exactement comme leurs homonymes `Serial.print` et `Serial.println` vues au § 2.9 auquel nous vous prions de bien vouloir vous reporter pour leur description.

- **write ()**

  Cette fonction permet de faire sortir sur la liaison série des données sous forme brute. Elle fonctionne exactement comme son homonyme `Serial.write` vue au § 2.9 à laquelle nous vous prions de bien vouloir vous reporter pour sa description.

6.1.2 L'interface RS 232

Les signaux logiques aux niveaux TTL ou CMOS, générés par les circuits logiques dont les microcontrôleurs font partie, acceptent assez mal de voyager sur plus de quelques centimètres car leurs formes se dégradent alors à un point tel que leur

exploitation devient impossible. Pour établir une liaison série sur une distance raisonnable, allant de quelques dizaines de centimètres à plusieurs centaines de mètres, diverses normes ont donc vu le jour dont la célèbre norme RS 232.

Cette dernière, appelée aussi CCITT V24 ou encore V24 « tout court », est d'origine américaine et définit deux choses : les niveaux électriques des signaux utilisés pour la transmission mais aussi un certain nombre de lignes, autres que les lignes d'émission et de réception de données, ayant des fonctions de contrôle. Cette norme est très précise et, théoriquement, il est possible de connecter directement et sans problème deux équipements qui la respectent ; malheureusement ce n'est pas toujours le cas et de nombreux appareils ne suivent qu'une partie de la norme.

Au point de vue niveau, cette norme est très simple : tout signal de niveau compris entre +3 et +25 volts est considéré comme étant au niveau logique A alors que tout signal compris entre –3 et –25 volts est considéré comme étant au niveau logique B. A et B sont quelconques et peuvent être 0 ou 1 selon que l'on travaille en logique positive ou négative.

Si donc vous voulez relier votre Arduino à un équipement doté d'une interface série à la norme RS 232, il va impérativement falloir faire appel à une interface car, dans le cas contraire, en sortie de l'Arduino cela ne fonctionnera pas (niveaux CMOS insuffisants pour la norme RS 232) et, en entrée de l'Arduino, celui-ci sera détruit très rapidement en raison des niveaux RS 232 d'amplitude beaucoup trop importante pour ce que peuvent supporter ses entrées.

Disons le tout net, nous ne comprenons pas pourquoi on voit encore aujourd'hui tant de circuits d'interface RS 232 plus ou moins bricolés, ou faisant appel à des tensions d'alimentation multiples alors que des circuits, peu coûteux et très répandus, permettent de réaliser une interface RS 232 parfaite (au plan normatif) avec très peu, voire même pas, de composants externes.

Le circuit le plus connu est sans conteste le MAX 232 de Maxim, repris depuis par de nombreux autres fabricants en seconde source. Ce circuit dispose en interne de deux interfaces TTL vers RS 232 et des deux interfaces inverses, RS 232 vers TTL. Il dispose aussi, et c'est ce qui fait son intérêt, de deux convertisseurs de tension statiques à capacités commutées qui lui permettent de générer de vrais niveaux RS 232 bipolaires (+ et –9 volts) avec une seule alimentation 5 volts. Pour cela, comme le montre la figure 6.1, il ne lui faut que quatre condensateurs externes de faibles valeurs.

L'investissement à prévoir est dérisoire et permet de disposer de véritables niveaux RS 232 compatibles de toute interface à cette norme, sans aucune restriction.

Si votre application est confrontée à des problèmes de place et que les quatre condensateurs nécessaires au MAX 232 sont trop encombrants, vous pouvez faire appel au MAX 233 dont le schéma de mise en œuvre est présenté figure 6.2.

Ce circuit fonctionne comme le MAX 232 mais n'a plus besoin d'aucun condensateur externe. Et si la version en boîtier DIL 20 pattes est trop encombrante, sachez qu'il existe aussi en boîtier à montage en surface de type SO20.

© Dunod – La photocopie non autorisée est un délit.

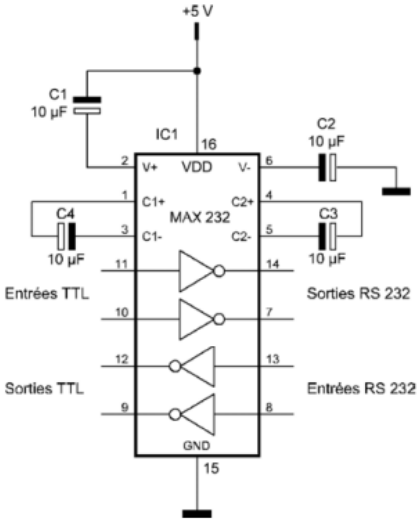

Figure 6.1 – Utilisation du MAX 232
pour disposer d'une véritable interface RS 232.

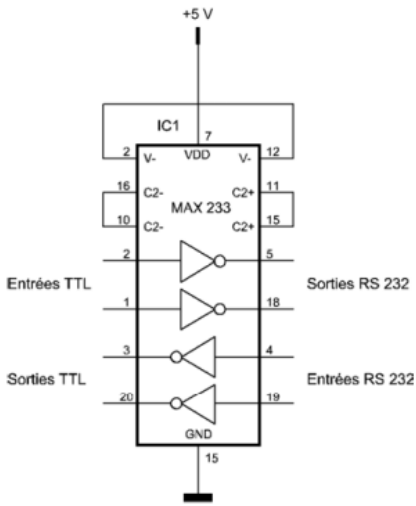

Figure 6.2 – Le MAX 233 n'a besoin d'aucun condensateur externe.

D'autres versions de circuits de ce type sont proposées par Maxim puisque les références de ces produits s'étagent du MAX 220 au MAX 249. Elles nous semblent moins bien adaptées aux besoins d'une application à base d'Arduino. Vous trouverez cependant les fiches techniques de tous les circuits de cette série dans les compléments en ligne de cet ouvrage.

6.1.3 Des signaux de contrôle parfois inutiles

Dans le cadre de cet ouvrage, nous n'allons pas procéder à une étude détaillée de la norme RS 232 mais sachez tout de même que, outre les signaux de données RxD et TxD, elle définit aussi un certain nombre de signaux de contrôle que vous êtes libre d'utiliser ou non. Par contre, si vous connectez une liaison série minimale, c'est-à-dire ne comportant que TxD, RxD et la masse, à un équipement « complet » c'est-à-dire gérant les signaux de contrôle, cela ne fonctionnera pas car ce dernier attendra en permanence le passage des lignes de contrôle aux niveaux requis pour que la transmission puisse avoir lieu. Ce problème se produit en particulier lorsque l'on connecte un Arduino à un PC (autrement que via le port USB de l'environnement de développement bien sûr) car l'interface série COMx du PC gère les signaux de contrôle.

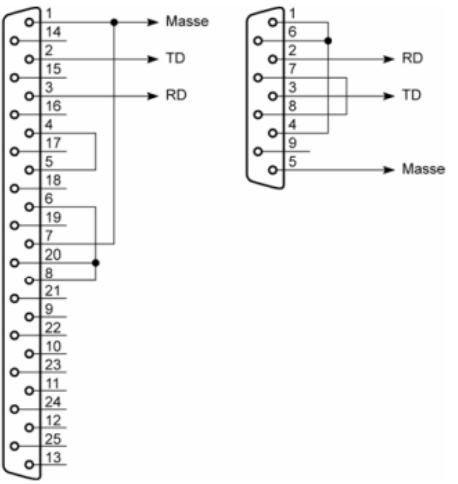

Figure 6.3 – Les straps à réaliser pour tromper les signaux de contrôle d'une liaison RS 232.

© Dunod – La photocopie non autorisée est un délit.

Pour relier votre Arduino à un équipement qui utilise une liaison RS 232 complète, il faut donc « tromper » ce dernier au moyen d'un bouchon de court-circuit. Un tel bouchon boucle les sorties des signaux de contrôle sur les entrées correspondantes faisant ainsi croire à l'équipement que l'Arduino auquel il est connecté gère les signaux de contrôle et qu'il est prêt. La figure 6.3 présente les schémas de tels bouchons dans les cas des prises normalisées à 25 points et à 9 points utilisées sur les liaisons séries RS 232.

6.2 Les liaisons séries synchrones

La majorité des liaisons que nous allons étudier dans la suite de ce chapitre sont des liaisons séries synchrones c'est-à-dire que, contrairement à la liaison série asynchrone que nous venons de présenter, elles nécessitent toutes la présence d'un signal d'horloge conjointement au signal véhiculant les données.

Cependant, un certain nombre de ces liaisons ayant fait l'objet d'une normalisation (I2C, SPI, One Wire, etc.) nous avons décidé de traiter séparément la liaison série synchrone, que vous pouvez réaliser vous-même avec un banal registre à décalage et votre Arduino, de ces liaisons normalisées que nous verrons donc dans la suite de ce chapitre.

Voici donc tout d'abord comment réaliser une liaison série synchrone avec un registre à décalage ou, plus prosaïquement car c'est souvent à cela que ça sert, comment économiser des lignes d'entrées/sorties parallèles grâce à un registre à décalage.

6.2.1 Le registre à décalage

Même s'il existe aujourd'hui une multitude de références de registres à décalage, le plus répandu, et donc le plus utilisé avec l'Arduino, est le 74HC595 dont le brochage et le synoptique sont représentés figure 6.4.

Comme le montre sa table de vérité visible tableau 6.1, un tel circuit reçoit sur son entrée DATA (appelée aussi DS selon les fiches techniques), des données sous forme série, c'est-à-dire encore des données placées les unes à la suite des autres. À chaque front montant du signal appliqué sur l'entrée CLOCK (appelée aussi SH_CP ou SHIFT_CLOCK), entrée qui est aussi appelée horloge, la donnée présente sur DATA est placée dans un étage du registre et tout le contenu de celui-ci est décalé vers la droite d'une position.

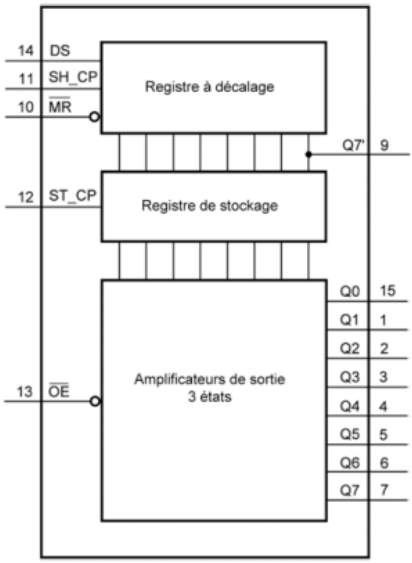

Figure 6.4 – Brochage et synoptique du registre à décalage 74HC595.

Tableau 6.1 – Table de vérité du registre à décalage 74HC595.

CLOCK SH_CP	LATCH ST_CP	/OE	/MR	DATA DS	Action
X	X	0	0	X	RAZ du registre à décalage
X	FM	0	0	X	Transfert du registre à décalage mis à zéro dans le registre de sortie
X	X	1	0	X	Sorties en mode haute impédance
FM	X	0	1	0 ou 1	Décalage d'une position du registre à décalage et entrée de DS dans le registre
X	FM	0	1	X	Transfert du registre à décalage dans le registre de sortie
FM	FM	0	1	X	Décalage d'une position et transfert du contenu précédent du registre à décalage dans le registre de sortie

© Dunod – La photocopie non autorisée est un délit.

Si l'on envoie sur DATA un octet sous forme série, il faudra donc huit fronts montants sur CLOCK pour disposer, dans le registre, de la donnée correspondante sous forme parallèle.

Pour que cette donnée soit disponible sur les sorties QA à QH (appelées aussi Q0 à Q7), il faudra tout à la fois que /G (appelée aussi /OE) soit au niveau bas, ce qui valide les sorties et qu'au moins une impulsion montante soit envoyée sur LATCH (appelée aussi ST_CP). C'est en effet cette dernière qui fait transférer les données reçues en interne sur les sorties QA à QH.

Le circuit dispose également d'une entrée de reset, appelée /MR qui, lorsqu'elle est mise à 0, réinitialise seulement les données reçues en interne mais pas l'état courant des sorties.

Dans la pratique, et lorsque l'on fait appel à un tel registre avec un Arduino, c'est généralement pour économiser des lignes de sorties parallèles, soit pour commander de nombreuses LED, soit pour commander un ou plusieurs afficheurs sept segments (qui ne sont que des assemblages de LED rappelons-le !). Voici comment l'on procède.

6.2.2 Commande de LED ou d'afficheurs avec un registre à décalage

Nous allons prendre à titre d'exemple la commande d'un afficheur sept segments à LED qu'il vous sera ensuite facile de transposer si nécessaire à de la commande de LED indépendantes puisque le principe sera le même.

Notre afficheur et son registre peuvent être câblés par exemple comme indiqué figure 6.5. Notez toutefois que cela n'a rien d'impératif. N'importe quelles sorties numériques de l'Arduino peuvent être utilisées pour commander DATA, CLOCK et LATCH de même que la répartition des segments de l'afficheur sur les sorties Q0 à Q7 peut être différente de celle proposée ici. Il conviendra alors seulement de modifier la table de transcodage présente dans le programme de gestion en conséquence.

En fait, et comme nous pouvons faire ce que nous voulons, le schéma de câblage adopté sur cette figure 6.5 correspond à celui du Danger Shield présenté au chapitre 3. Si vous en disposez, vous pourrez ainsi essayer le programme présenté ci-après sans avoir à réaliser le moindre câblage.

À la lecture de la table de vérité du 74HC595 ou pourrait s'attendre à ce que la commande du 74HC595 par l'Arduino soit difficile. En fait il n'en est rien grâce à la présence de la fonction très puissante qu'est `shiftOut`. Nous vous renvoyons si nécessaire à sa description au § 2.7 avant de lire le programme ci-dessous qui fait afficher tous les chiffres de 0 à 9 sur l'afficheur câblé comme indiqué figure 6.5.

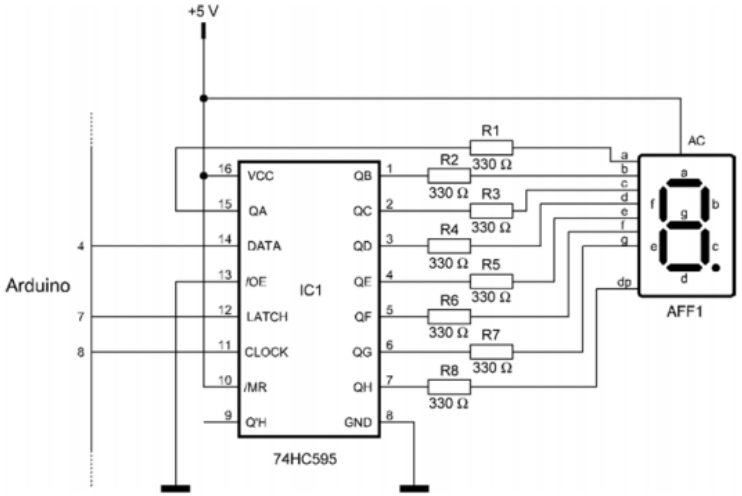

Figure 6.5 – Exemple d'utilisation d'un 74HC595 pour commander un afficheur sept segments à LED.

Listing 6.1 – Exemple de gestion d'un afficheur sept segments à LED connecté au moyen d'un registre à décalage.

```
// Définition connexion 74HC595 sur Arduino

#define LATCH 7
#define CLOCK 8
#define DATA 4

// Définition des segments à allumer pour
// afficher les chiffres de 0 à 9

const byte ledCharSet[10] = {
  B00111111,B00000110,B01011011,
  B01001111,B01100110,B01101101,
  B01111101,B00000111,B01111111,B01101111
};

int chiffre = 0;

void setup()
{
  pinMode(LATCH, OUTPUT);
  pinMode(CLOCK, OUTPUT);
  pinMode(DATA,OUTPUT);
}
```

© Dunod – La photocopie non autorisée est un délit.

```
void loop()
{
  for (chiffre = 0; chiffre <10; chiffre++)
  {
    digitalWrite(LATCH,LOW);
// Blocage des sorties du 74HC595
    shiftOut(DATA,CLOCK,MSBFIRST,~(ledCharSet[chiffre]));
// Décalage d'un chiffre
    digitalWrite(LATCH,HIGH);
// Transfert du chiffre décalé en sortie
    delay(500) ;
  }
}
```

Paradoxalement, la partie la plus importante de ce programme se situe presque au niveau des définitions et plus particulièrement de celle du tableau baptisé `ledCharSet`. C'est en effet à son niveau que sont définis les segments à allumer en fonction des chiffres à afficher, étant entendu que ces chiffres sont constitués par l'indice utilisé pour parcourir ce tableau. Ainsi, par exemple, `ledCharSet[2]` correspond-il au dessin du chiffre 2.

Ce tableau doit être rempli en tenant compte de deux paramètres. Le premier est le mode de raccordement des segments a à f de l'afficheur sur les sorties QA à QH du 74HC595. Il suffit donc d'examiner la figure 6.5 pour savoir ce qu'il en est. Le second est le mode d'utilisation de la fonction `shiftOut` qui est paramétrée ici ave `MSBFIRST` ce qui signifie qu'elle va faire générer les données émises sous forme série avec le bit de poids fort en tête, c'est-à-dire encore avec le bit qui se retrouvera sur QH en tête.

Ainsi par exemple, pour dessiner le chiffre 1, il faut allumer les segments b et c, c'est-à-dire encore envoyer 1 sur QB et 1 sur QD puisque QB est relié au segment b et QD au segment D. Il faut donc que `ledCharSet[1]` soit égal à `00000110` ce qui est bien le cas dans le listing ci-dessus. Nous vous laissons le soin de vérifier ce qu'il en est pour les autres chiffres afin, pourquoi pas, de définir l'affichage d'autres caractères ou symboles si vous le souhaitez.

Le reste du programme est très facile à comprendre dès lors que l'on a lu la description du 74HC595 faite précédemment. L'essentiel de la tâche de communication est en effet accompli par `shiftOut`. Notez à ce propos que le contenu de `ledCharSet` est inversé au niveau de cette fonction car l'afficheur utilisé sur le Danger Shield est un modèle à anodes communes et qu'il faut donc mettre les segments à 0 pour les allumer, alors que nous avons établi le contenu de `ledCharSet` comme s'il fallait les mettre à 1.

6.3 Le bus I2C

Le succès remporté par le bus I2C, très utilisé depuis déjà de nombreuses années dans les appareils audio et vidéo, a comme conséquence que de nombreux boîtiers sont disponibles avec une telle interface. On trouve ainsi des mémoires bien sûr

mais aussi des convertisseurs analogiques/digitaux et digitaux/analogiques, des horloges temps réel, des interfaces parallèles ainsi que toute une gamme de circuits aux fonctions beaucoup plus complexes puisqu'il existe même des synthétiseurs de fréquence intégrés pour récepteurs radio ou TV.

Même si c'est un bus série, qui pourrait donc s'accommoder des explications précédentes, il est nécessaire de vous donner quelques précisions supplémentaires à son sujet afin que vous puissiez l'utiliser au mieux. En effet, même si une bibliothèque spécialisée lui est consacrée afin de faciliter au maximum son utilisation, la connaissance des principes de base de son protocole s'avère quasiment indispensable à sa bonne mise en œuvre.

6.3.1 Généralités

Le bus I2C, qui n'utilise que deux lignes de signal (et les masses correspondantes bien sûr), permet d'échanger des informations sous forme série avec un débit pouvant atteindre 100 kilobits par seconde ou 400 kilobits par seconde pour les versions les plus récentes. Ses points forts sont les suivants :

- C'est un bus série bifilaire utilisant une ligne de données appelée SDA (*Serial DAta*) et une ligne d'horloge appelée SCL (*Serial CLock*).
- Les données peuvent être échangées dans les deux sens sans restriction.
- Le bus est multi maîtres.
- Chaque abonné dispose d'une adresse codée sur 7 bits. On peut donc connecter simultanément 128 abonnés d'adresses différentes sur le même bus (sous réserve de ne pas le surcharger électriquement bien sûr).
- Un acquittement est généré pour chaque octet de donnée transféré.
- Le bus peut travailler à une vitesse maximum de 100 kilobits par seconde ou 400 kilobits par seconde pour les versions les plus récentes, étant entendu qu'un procédé automatique permet de ralentir l'équipement le plus rapide pour s'adapter à la vitesse de l'élément le plus lent lors d'un transfert.
- Le nombre maximum d'abonnés n'est limité que par la charge capacitive maximale du bus qui peut être de 400 pF. Ce nombre ne dépend donc que de la technologie des circuits et du mode câblage employés.
- Les niveaux électriques permettent l'utilisation de circuits en technologies CMOS, NMOS ou TTL.

6.3.2 Protocole du bus I2C

La figure 6.6 montre le principe adopté au niveau des étages d'entrée/sortie des circuits d'interface au bus I2C. Si la partie entrée n'appelle aucune remarque particulière, on constate que la partie sortie fait appel à une configuration à drain ouvert (l'équivalent en MOS du classique collecteur ouvert) ce qui permet de réaliser des ET câblés par simple connexion, sur la ligne SDA ou SCL, des sorties de tous les circuits.

© Dunod – La photocopie non autorisée est un délit.

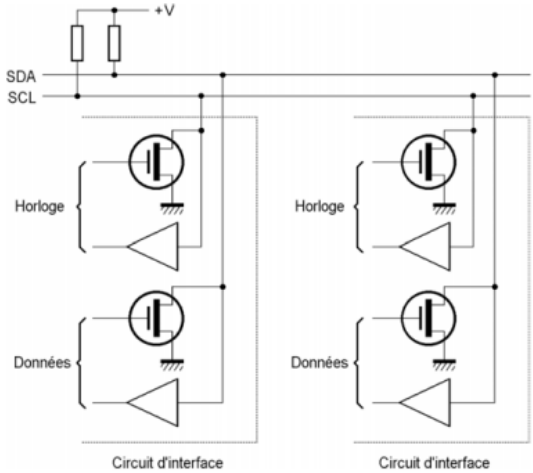

Figure 6.6 – Schéma de l'interface matérielle des circuits compatibles du bus I2C.

Aucune charge n'étant prévue dans ces derniers, une résistance de rappel à une tension positive doit être mise en place. Le niveau électrique n'est pas précisé pour l'instant car il dépend de cette tension. Nous parlerons donc de niveaux logiques hauts ou « 1 » ou bien encore de niveaux logiques bas ou « 0 » étant entendu que l'on travaille en logique positive c'est-à-dire qu'un niveau haut correspond à une tension plus élevée qu'un niveau bas.
Compte tenu de ce mode de connexion en ET câblé, lorsque aucun abonné n'émet sur le bus, les lignes SDA et SCL sont au niveau haut qui est leur état de repos.

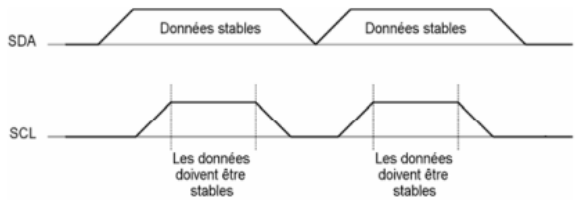

Figure 6.7 – Chronogramme fondamental du bus I2C.

Ceci étant vu, la figure 6.7 résume le principe fondamental d'un transfert à savoir : une donnée n'est considérée comme valide sur le bus que lorsque le signal SCL est à l'état haut. L'émetteur doit donc positionner la donnée à émettre lorsque SCL est à l'état bas et la maintenir tant que SCL reste à l'état haut.

Comme la transmission s'effectue sous forme série, une information de début et de fin doit être prévue. L'information de début s'appelle ici condition de départ et l'information de fin s'appelle condition d'arrêt.

Une condition de départ est réalisée lorsque la ligne SDA passe du niveau haut au niveau bas alors que SCL est au niveau haut. Réciproquement, une condition d'arrêt est réalisée lorsque SDA passe du niveau bas au niveau haut alors que SCL est au niveau haut.

En outre, bien que nous soyons en présence d'un bus série, les données sont envoyées par paquets de huit ou octets, même si un octet peut parfois regrouper huit bits indépendants. Le bit de poids fort est envoyé le premier. Chaque octet est suivi par un bit d'acquittement de la part du destinataire. L'ensemble du processus fonctionnant de la façon présentée figure 6.8.

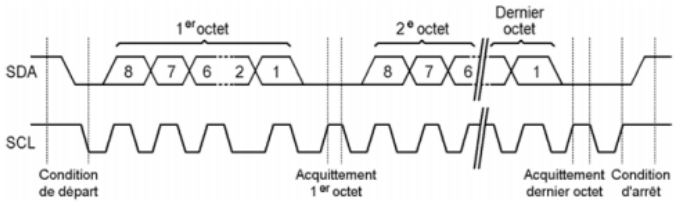

Figure 6.8 – Chronogramme d'un échange complet sur le bus I2C.

Tout d'abord, précisons que, lors d'un échange de ce type, la ligne SCL est pilotée par l'initiateur de l'échange ou maître, quitte à ce que l'esclave agisse également dessus dans certains cas particuliers.

La figure 6.8 nous montre tout d'abord une condition de départ, générée par le maître du bus à cet instant. Elle est suivie par le premier octet de données, poids forts en tête. Après le huitième bit, l'émetteur qui est aussi le maître dans ce cas, met sa ligne SDA au niveau haut c'est-à-dire au repos mais continue à générer l'horloge sur SCL. Pour acquitter l'octet, le récepteur doit alors forcer la ligne SDA au niveau bas pendant l'état haut de SCL qui correspond à cet acquittement, prenant en quelque sorte la place d'un neuvième bit.

Le processus peut alors continuer avec l'octet suivant et se répéter autant de fois que nécessaire pour réaliser un échange d'informations complet. Lorsque cet échange est terminé, le maître génère une condition d'arrêt.

Maintenant que nous savons comment se déroulent les échanges ; il nous reste à examiner le format des données transmises afin de comprendre comment fonctionnent l'adressage, mais aussi la définition du sens de transferts des données.

© Dunod – La photocopie non autorisée est un délit.

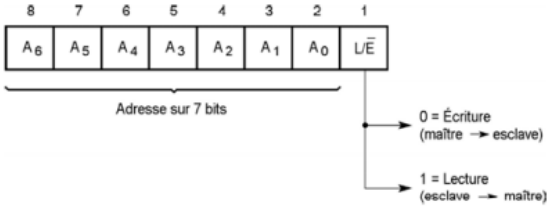

Figure 6.9 – Contenu de l'octet d'en-tête d'un échange sur le bus I2C.

La figure 6.9 montre le contenu du premier octet qui est toujours présent en début d'échange. Ses sept bits de poids forts contiennent l'adresse du destinataire du message, ce qui autorise 128 combinaisons différentes. Le bit de poids faible quant à lui indique si le maître va réaliser une lecture ou une écriture. En d'autres termes, si ce bit est à zéro le maître va écrire dans l'esclave ou lui envoyer des données. Si ce bit est à un, le maître va lire dans l'esclave c'est-à-dire que le maître va recevoir des données de l'esclave.

Lorsqu'un maître désire effectuer plusieurs échanges à destination d'esclaves d'adresses différentes, il n'est pas obligé de terminer le premier échange par une condition d'arrêt mais peut les enchaîner en générant une condition de départ dès la fin d'un échange.

Enfin il existe une procédure dite d'appel général où l'adresse envoyée par le maître, c'est-à-dire rappelons-le, les sept bits de poids forts du premier octet, est nulle. Tous les circuits connectés sur le bus capables de répondre à un tel appel général doivent alors le faire et prendre en compte les données qui suivent. Leur attitude dépend du bit de lecture/écriture de ce premier octet. En effet, si ce bit est à zéro, le deuxième octet revêt une signification particulière, que nous ne détaillerons pas ici, mais qui permet, au niveau de chaque esclave, la programmation de son adresse par logiciel ou matériel.

Lorsque ce bit est à un, on est en présence d'un appel général matériel qui est transmis par un maître du bus matériel, tel qu'un circuit d'interface de clavier par exemple. Un tel circuit est incapable de générer l'adresse du destinataire de l'information qu'il a à envoyer. Dans ces conditions, l'octet suivant contient l'adresse de ce maître matériel afin qu'un esclave « intelligent », généralement un microcontrôleur, le reconnaisse et prenne en compte l'information qu'il émet ensuite.

Il y aurait encore beaucoup à dire à propos du bus I2C mais ce n'est pas le but de cet ouvrage aussi avons nous restreint l'exposé précédent au strict minimum nécessaire afin que vous puissiez interfacer au mieux des circuits I2C avec l'Arduino au moyen de la bibliothèque prévue à cet effet.

6.3.3 La bibliothèque Wire de l'Arduino

La gestion du bus I2C avec l'Arduino est relativement facile à mettre en œuvre pour deux raisons. La première est que les microcontrôleurs Atmel ATmega qui

l'équipent disposent d'origine d'une interface I2C. Les deux lignes SDA et SCL sont ainsi accessibles respectivement sur A4 et A5 sur les Uno et sur 20 et 21 sur le Mega 2560. Bien évidemment, dans le cas du Uno, les entrées analogiques A4 et A5 deviennent inutilisables lorsqu'elles sont affectées au bus I2C.

> **Attention !**
> À partir des Arduino Uno Révision 3, ces deux lignes SDA et SCL sont également reportées à l'extrémité du connecteur situé à côté du poussoir de reset. Cela ne signifie pas pour autant qu'elles deviennent indépendantes de A4 et A5 avec lesquelles elles sont toujours communes du fait de la structure interne de l'ATmega 328.

La deuxième raison est la présence d'origine, dans l'environnement de développement, d'une bibliothèque spécialisée baptisée `Wire.h` qui permet une gestion logicielle aussi simple que possible du protocole, tout de même assez particulier, du bus I2C. En voici les fonctions principales.

■ begin()

Cette fonction, qui peut s'utiliser sous deux formes différentes, doit être appelée une fois avant toute utilisation de la bibliothèque. Elle initialise en effet le bus I2C et y raccorde un maître ou un esclave.

```
Wire.begin() ;
Wire.begin(adresse) ;
```

Dans le premier cas, l'Arduino est raccordé au bus I2C en tant que maître du bus, dans le second cas en tant qu'esclave avec l'adresse qui est alors spécifiée.

■ beginTransmission()

Cette fonction doit être appelée avant tout échange du maître vers l'esclave. Elle initie la transmission à destination de l'esclave dont l'adresse est spécifiée de la façon suivante :

```
Wire.beginTransmission(adresse) ;
```

Concrètement, cette fonction génère en fait une condition de départ à destination de l'esclave ainsi désigné. Elle doit donc être suivie par un envoi de données.

■ write(), anciennement send()

Sur le maître, cette fonction assure l'envoi vers l'esclave préalablement désigné d'une ou plusieurs données. Sur l'esclave, elle assure l'envoi de données préalablement demandées par le maître. Elle peut s'utiliser sous trois formes différentes :

```
Wire.write(valeur) ;
Wire.write(chaine de caractères) ;
Wire.write(donnée, nombre) ;
```

© Dunod – La photocopie non autorisée est un délit.

Dans le premier cas, `valeur` est la variable de type `byte` qui est transmise. Dans le second cas, `chaine de caractères` est un pointeur de type `byte` vers une chaine de caractères à transmettre. Dans le troisième cas enfin, `donnée` est un pointeur de type `byte` vers un tableau tandis que `nombre`, également de type `byte`, indique le nombre d'éléments de ce tableau à transmettre.

■ requestFrom()

Cette fonction est en quelque sorte le complément de la précédente. Elle permet au maître de demander des données à un esclave et s'utilise sous la forme :

```
Wire.requestFrom(adresse, nombre) ;
```

`adresse` est évidemment l'adresse de l'esclave auquel sont demandées les données tandis que `nombre` indique le nombre d'octets demandés. Les octets qui seront envoyés par l'esclave suite à cette demande pourront être lus au moyen des fonctions `available` et `read` présentées ci-dessous.

■ available()

Cette fonction, dont l'utilisation n'est pas obligatoire, permet de savoir combien d'octets sont disponibles à la lecture au moyen d'une fonction `read`, suite à l'exécution d'une fonction `requestFrom`. Elle s'utilise tout simplement sous la forme :

```
nombre = Wire.available() ;
```

`nombre` est une variable, au moins de type `byte`, contenant le nombre d'octets disponibles.

■ read(), anciennement receive()

Sur un esclave, cette fonction permet de lire un octet transmis par le maître tandis que, sur un maître, elle permet de lire un octet transmis pas un esclave suite à l'exécution d'une fonction `requestFrom`. Sa syntaxe est la suivante :

```
donnée = Wire.read() ;
```

où `donnée` est une variable au moins de type `byte` qui contient le caractère lu sur le bus I2C.

■ endTransmission()

Comme son nom l'indique cette fonction met fin à une séquence de transmission préalablement initié avec la fonction `beginTransmission`. Elle renvoie en retour une donnée de type `byte` dont le contenu peut évoluer de 0 à 4 afin d'indiquer l'état de la transmission de la façon suivante :

– 0 signifie que l'échange s'est bien déroulé ;
– 1 indique que les données à transmettre étaient trop nombreuses pour tenir dans le tampon d'émission ;

– 2 indique qu'un NACK (absence d'acquittement) a été reçu suite à l'envoi de l'adresse ;
– 3 indique qu'un NACK (absence d'acquittement) a été reçu suite à l'envoi d'une donnée ;
– 4 indique une erreur d'un autre type.

La syntaxe habituelle de cette fonction est la suivante :

```
code = Wire.endTransmission() ;
```

où `code` est une variable de type `byte` qui contient l'un des résultats ci-dessus.
Cette fonction peut également s'utiliser avec un paramètre optionnel sous la forme :

```
code = Wire.endTransmission(stop) ;
```

Le paramètre `stop`, qui est nécessairement de type boolean, peut alors être égal à un pour générer une condition d'arrêt sur la liaison I2C ou à zéro pour générer une condition de départ répété.

6.3.4 Exemple d'utilisation du bus I2C

Si vous n'avez pas l'habitude d'utiliser le bus I2C, tout ceci vous laisse peut-être un peu perplexe aussi allons-nous voir maintenant deux exemples. L'Arduino sera utilisé comme maître du bus I2C dans les deux cas, ce qui est généralement son rôle dans la majorité des applications mais, dans le premier exemple nous ne ferons qu'envoyer des données à un esclave alors que, dans le deuxième exemple, nous enverrons et recevrons des données de l'esclave.

Avant de voir les schémas et programmes correspondants, précisons toutefois comment l'on doit travailler lorsque l'on veut interfacer un circuit I2C avec l'Arduino. La méthodologie est en effet toujours la même.

La première opération, qui est bien souvent la plus longue et la plus délicate, consiste à lire en détail la fiche technique du circuit afin de prendre connaissance de sa ou de ses adresses, des registres qu'il contient et de ce qu'il faut y écrire pour réaliser les fonctions désirées.

Une fois que c'est fait, et éventuellement résumé noir sur blanc sur une feuille, il ne reste plus qu'à enchaîner les fonctions que nous venons de décrire pour envoyer et/ou recevoir les données correspondantes. En voici un premier exemple avec un potentiomètre numérique.

Le schéma de notre montage est présenté figure 6.10 et met en œuvre un potentiomètre numérique commandé par un bus I2C ; en l'occurrence un AD5171 d'Analog Devices dont vous trouverez la fiche technique dans les compléments en ligne.

Le dialogue avec un tel potentiomètre est fort simple puisque, comme l'indique sa fiche technique, il est à l'adresse fixe 0x2C (soit 44 en décimal) lorsque l'on relie son entrée AD0 à la masse, et qu'il ne comporte que deux registres.

© Dunod – La photocopie non autorisée est un délit.

Dans le premier on doit écrire la commande désirée, qui se limitera ici à 00 puisque nous laisserons volontairement de côté dans cet exemple le mode OTP de ce potentiomètre, qui permet de le programmer sur une position une fois pour toutes. Le second registre quant à lui sert à écrire la position relative du potentiomètre sous la forme d'une donnée comprise entre 0 et 63 puisque le potentiomètre comporte 64 pas.

Notez que nous n'utiliserons pas ici la fonction de lecture qui permet de savoir dans quelle position il se trouve. Dans ces conditions, le programme de gestion est fort simple comme vous pouvez le découvrir ci-après.

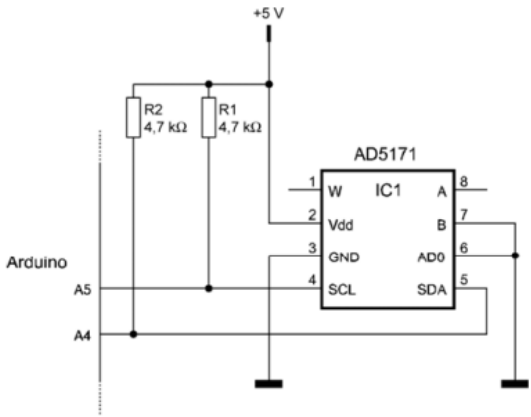

Figure 6.10 – Commande d'un potentiomètre numérique à interface I2C.

Listing 6.2 – Exemple de gestion d'un potentiomètre numérique à interface I2C.

```
#include <Wire.h>

void setup()
{
  Wire.begin();  // Initialisation I2C, Arduino en maître
}

byte position = 0;

void loop()
{
  Wire.beginTransmission(0x2C); // L'AD5171 est à l'adresse 0x2C
  Wire.write(0x00);             // Envoi de l'instruction
  Wire.write(position);         // Envoi de la position
  Wire.endTransmission();       // Fin de l'échange
```

```
    position++;
    if(position == 64)  // Si le maximum est atteint
    {
      position = 0;     // Retour au minimum
    }
    delay(500);
}
```

Ce listing se laisse très facilement analyser. On commence en effet par initier le bus I2C avec l'Arduino en maître au moyen de `Wire.begin`. L'échange sur le bus I2C, qui consiste donc à écrire dans le circuit la commande (00 dans ce cas) suivie de la position du potentiomètre, contenue dans la variable `position`, se déroule ensuite comme nous l'a enseigné la description que nous avons faite du protocole I2C avec, tout d'abord, une condition de départ et la génération de l'adresse du circuit au moyen de `Wire.beginTransmission`, suivie de l'envoi des deux octets l'un après l'autre grâce à `Wire.write`. Le tout est terminé par une condition d'arrêt générée par `Wire.endTransmission`.

Cet exemple n'ayant fait qu'écrire dans un circuit I2C, nous vous proposons maintenant une deuxième application dans laquelle nous allons écrire mais aussi lire dans un tel circuit. En outre, il s'agit d'une application que vous aurez peut-être l'occasion d'utiliser car elle fait à appel à une horloge temps réel sauvegardée par batterie, utilisant le circuit aujourd'hui très répandu DS 1307 dont vous trouverez la fiche technique dans les compléments en ligne.

La figure 6.11 présente un schéma de mise en œuvre type, que vous pourrez par exemple câbler sur un shield prototype, encore que de nombreux modules prêts à l'emploi équipés du DS 1307 et de sa batterie de sauvegarde soient disponibles sur Internet.

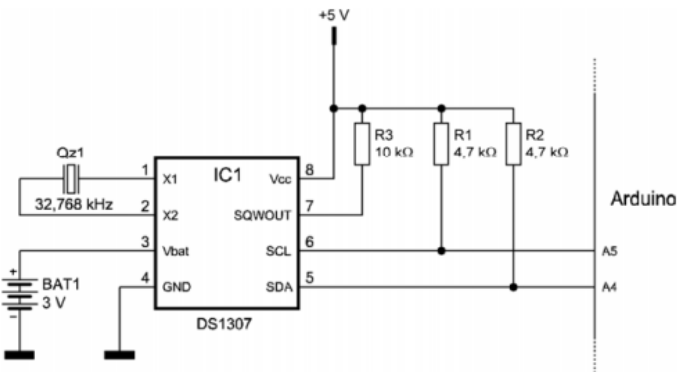

Figure 6.11 – Utilisation d'une horloge temps réel DS 1307 à interface I2C.

© Dunod – La photocopie non autorisée est un délit.

Comme pour le circuit précédent, la première tâche à accomplir consiste à lire la fiche technique du DS 1307 afin de prendre connaissance de ses registres et de leur contenu. Comme vous pourrez le constater, et si l'on ignore la RAM que contient également ce circuit, la partie horloge est contrôlée par huit registres servant à définir l'heure et la date complètes (des secondes aux années !) ainsi que la gestion de la sortie SQWOUT sur laquelle on peut faire générer un signal carré.

Ces registres sont accessibles en lecture et en écriture, c'est-à-dire que c'est là que l'on va venir lire l'heure mais c'est également dans ces mêmes registres que l'on va venir écrire pour mettre à l'heure l'horloge lors de sa première mise sous tension.

L'intérêt de notre exemple étant de montrer comment lire et écrire dans un circuit à interface I2C, nous n'avons pas particulièrement développé la partie mise à l'heure de notre programme. Celle-ci est en effet fixée par des constantes qu'il vous appartient d'initialiser avant de le charger dans l'Arduino.

De même, l'affichage de l'heure a lieu via le moniteur sur port série intégré à l'environnement de développement. Ceci étant, vous pouvez reprendre intégralement la partie gestion de l'horloge qu'il contient et en faire un simple copier-coller dans le programme de votre application, dans lequel vous pourrez inclure, par exemple, un affichage LCD ou une mise à l'heure au moyen d'un clavier ou de poussoirs en utilisant les exemples du chapitre 4.

Listing 6.3 – Exemple de gestion d'une horloge DS 1307 à interface I2C.

```
#include <Wire.h>

void setup()
{
  Wire.begin();  // Initialisation Arduino en mode maître
  Serial.begin(9600);
  MiseALHeure();  // Mise à l'heure de l'horloge
}

void loop()
{

  // Initialisation du pointeur de registre

  Wire.beginTransmission(0x68); // Conditions de départ
  Wire.write(0);                // Écriture du pointeur de registre
  Wire.endTransmission();       // Condition d'arrêt

  // Lecture des 7 octets contenant date et heure

  Wire.requestFrom(0x68, 7);
  byte secs = Wire.read();
  byte mins = Wire.read();
  byte hrs = Wire.read();
  byte jour = Wire.read();
  byte date = Wire.read();
  byte mois = Wire.read();
  byte annee = Wire.read();
```

```
  // Mise en forme et affichage des données

  Serial.print("Heure ");
  if (hrs < 10) Serial.print("0");  // Ajout d'un 0 en tête
  Serial.print(hrs,HEX);
  Serial.print(":");
  if (mins < 10) Serial.print("0"); // Ajout d'un 0 en tête
  Serial.print(mins, HEX);
  Serial.print(":");
  if (secs < 10) Serial.print("0"); // Ajout d'un 0 en tête
  Serial.println(secs, HEX);
  Serial.print("Date ");
  if (date < 10) Serial.print("0"); // Ajout d'un 0 en tête
  Serial.print(date,HEX);
  Serial.print("-");
  if (mois < 10) Serial.print("0"); // Ajout d'un 0 en tête
  Serial.print(mois, HEX);
  Serial.print("-");
  Serial.print("20");               // Ajout de 20 pour les années
                                     ➥ 2000
  if (annee < 10) Serial.print("0");
  Serial.println(annee, HEX);
  Serial.println();
  delay(5000);
}

// Fonction de mise à l'heure simplifiée

void MiseALHeure()
{
  Wire.beginTransmission(0x68);  // Condition de départ
  Wire.write(0);                 // Initialisation pointeur
                                    ➥ de registres
  Wire.write(0x00);              // Secondes
  Wire.write(0x00);              // Minutes
  Wire.write(0x80 | 0x12);       // Heures en mode 24 H
  Wire.write(0x05);              // Jour 01-07
  Wire.write(0x20);              // Date 0-31
  Wire.write(0x01);              // Mois 0-12
  Wire.write(0x11);              // Année 00-99
  Wire.write(0x10);              // Gestion de la sortie SQWOUT
  Wire.endTransmission();        // Condition d'arrêt
}
```

L'analyse du programme est ici encore très facile à réaliser. Après avoir initialisé le bus I2C avec l'Arduino en mode maître, on appelle la fonction `MiseALHeure` qui se charge d'écrire l'heure et la date désirées dans les différents registres du DS1307. Ces écritures font appel aux mêmes opérations que celles vues ci-avant pour le potentiomètre numérique et nous n'y reviendrons donc pas.

La fonction `loop` quant à elle se charge de lire l'heure et la date dans le DS 1307 et de les faire afficher sur le moniteur de l'environnement de développement. Pour cela, elle commence par écrire 0 dans le DS 1307 afin de réinitialiser le pointeur d'adresses de ses registres. Ensuite, une fonction `requestFrom` est utilisée avec un

© Dunod – La photocopie non autorisée est un délit.

nombre d'octets à recevoir fixé à 7 puisque nous ne souhaitons lire que les registres d'heure et de date. Ces données peuvent alors être lues avec autant de fonction `read` que nécessaire et sont placées dans les variables correspondantes.

Comme vous pouvez le constater, que ce soit pour lire ou écrire sur le bus I2C, la bibliothèque `Wire` facilite grandement le travail.

6.4 L'interface ou bus SPI

Au royaume des circuits d'interface le bus I2C n'est pas seul et se trouve talonné depuis de nombreuses années par les circuits dotés d'une liaison série synchrone de type SPI. Cette appellation recouvre cependant des signaux et des chronogrammes qui sont beaucoup moins bien normalisés que ceux du bus I2C, tout simplement parce que la liaison SPI ne fait l'objet d'aucune norme officielle.

Certains circuits indiquent donc qu'ils sont munis d'une interface SPI, mais vous pourrez aussi trouver comme appellation Microwire (qui est une marque déposée de National Semiconductor) ou bien encore bus série trois fils. En effet, un bus de ce type permet la connexion, sous forme série maître-esclave, de plusieurs circuits disposant d'interfaces compatibles, avec seulement trois fils de liaison.

6.4.1 Principe d'une liaison de type SPI

La figure 6.12 présente le principe général d'interconnexion retenu. Ce schéma ne doit cependant pas être considéré comme unique. On peut en effet relier le SPI à de très nombreux autres circuits : registres à décalage, afficheurs disposant de leur propre logique (c'est ce qui sera fait ci-après à titre d'exemple), etc. Il suffit de s'assurer qu'ils sont à même de dialoguer avec l'interface, ce qui est une contrainte relativement simple à satisfaire vu la souplesse de programmation permise.

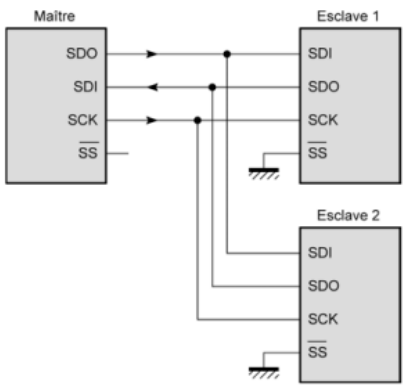

Figure 6.12 – Principe général d'une liaison de type SPI.

Si l'on reprend la figure 6.12 comme exemple, on y voit trois lignes « utiles » que sont SDO, SDI et SCK et une ligne de sélection de mode de fonctionnement du circuit concerné appelée /SS.

– La ligne SDO, pour *Serial Data Out*, est la sortie de données séries. Elle est aussi appelée MOSI ce qui signifie *Master Out Slave In*.
– La ligne SDI, pour *Serial Data In*, est comme son nom le laisse supposer, l'entrée de données série. Elle est aussi appelée MISO ce qui signifie *Master In Slave Out*.
– La ligne SCK pour *Serial ClocK* est utilisée pour synchroniser les échanges de données puisque nous sommes ici en mode série synchrone ne l'oublions pas. Cette ligne est nécessairement une sortie sur le maître et une entrée sur le ou les esclaves. La figure 6.13 montre les relations de phase qui existent entre les données et cette horloge étant entendu que celles-ci peuvent varier d'un circuit à un autre mais sont généralement programmables au niveau de l'élément « intelligent » qui gère le bus. Ce sera évidemment le cas de l'Arduino.
– La ligne /SS est la sélection du mode maître ou esclave. Elle est au niveau haut sur un maître et au niveau bas sur un esclave. Pour éviter toute erreur de transmission, cette ligne doit être au niveau bas sur le ou les esclaves avant toute transmission et y rester pendant toute la durée de celle-ci. En outre, il est évident qu'à un instant donné, n'importe quel circuit peut être maître et n'importe quel autre esclave, mais qu'il ne doit y avoir qu'un maître par transaction.

Toutes les liaisons SPI respectent le chronogramme de principe présenté figure 6.13 mais quatre variantes existent au niveau de la prise en compte des données qui sont échangées.

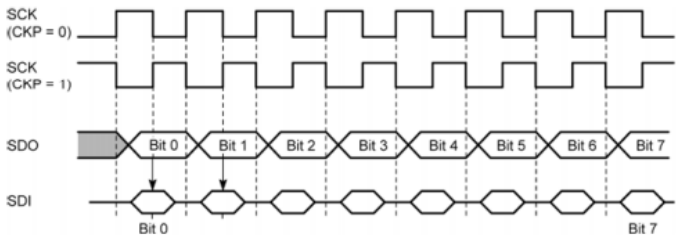

Figure 6.13 – Chronogramme général d'un échange sur une liaison SPI.

Ces variantes sont parfois appelées modes SPI et sont alors numérotées de 0 à 3. Nous préférons quant à nous dessiner un petit chronogramme permettant de bien préciser ce qu'il en est. En présence d'un circuit à interface SPI, il vous suffira donc de comparer sa fiche technique avec ce chronogramme pour savoir dans quel « mode » il fonctionne.

En fait, ces quatre modes sont définis par la polarité de l'horloge, appelée CPOL, qui peut être égale à 0 ou à 1, et par la phase de cette même horloge, appelée

© Dunod – La photocopie non autorisée est un délit.

CPHA, par rapport aux données, qui peut aussi être égale à 0 ou à 1. Cela conduit aux quatre possibilités visibles figure 6.14 qui indiquent en fait sur quel front du signal d'horloge les données sont échantillonnées et peuvent donc être considérées comme valides.

Par défaut, le mode le plus courant est le mode appelé mode 0 c'est-à-dire celui ou CPOL et CPHA sont tous deux nuls.

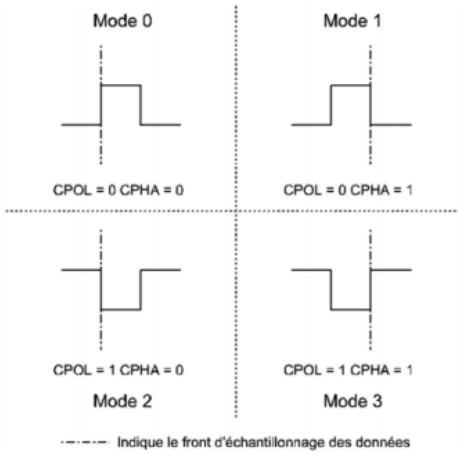

Figure 6.14 – Les quatre modes de prise en compte des données de la liaison SPI.

6.4.2 La bibliothèque SPI de l'Arduino

La gestion d'une liaison SPI avec l'Arduino est facile à mettre en œuvre pour les deux mêmes raisons que celles déjà vues pour le bus I2C. En effet, les microcontrôleurs Atmel ATmega qui l'équipent disposent d'origine d'une interface SPI dont les entrées/sorties se répartissent comme suit : SCK sur 13, MISO sur 12, MOSI sur 11 et /SS sur 10 pour les Uno et SCK sur 52, MISO sur 50, MOSI sur 51 et /SS sur 53 pour le Mega 2560.

L'environnement de développement contient, ici aussi, une bibliothèque spécialisée, baptisée `SPI.h`, qui permet une gestion logicielle très simple du protocole de dialogue des liaisons SPI. En voici les fonctions principales.

■ begin()

Cette fonction doit être appelée une fois avant toute utilisation de la liaison SPI de l'Arduino qu'elle sert à initialiser. Elle s'utilise tout simplement sous la forme :

```
SPI.begin() ;
```

■ **setDataMode(), anciennement mode()**

Cette fonction reste d'usage assez peu courant car elle sert à modifier le mode de fonctionnement de l'interface SPI de l'Arduino au cas où le paramétrage réalisé par défaut suite à l'utilisation de la fonction `begin` ne conviendrait pas. Celui-ci programme cette interface pour fonctionner en mode 0, c'est-à-dire avec CPOL à 0 et CPHA à 0. S'il s'avère nécessaire de modifier l'un ou l'autre, il faut appeler cette fonction de la façon suivante :

```
SPI.setDataMode(mode) ;
```

où mode peut être égal à SPI_MODE0, SPI_MODE1, SPI_MODE2 ou SPI_MODE3, chacun de ces modes correspondant aux diverses possibilités présentées figure 6.14.

■ **transfer()**

Cette fonction est évidemment la plus importante puisque c'est elle qui assure le transfert effectif des données sur la liaison SPI. Elle peut s'utiliser avec deux syntaxes différentes :

```
SPI.transfer(donnéeémise) ;
donnéereçue = SPI.transfer(donnéeémise) ;
```

Dans les deux cas, `donnéeémise` est une variable de type `byte` contenant la donnée à émettre sur la liaison SPI, de l'Arduino vers le circuit « esclave ».
Dans le dernier cas, le circuit connecté à l'Arduino sur la liaison SPI renvoie une donnée en retour ; donnée que l'on retrouve donc dans la variable `donnéereçue`.

■ **end()**

Même si ce n'est pas indispensable, il est conseillé d'appeler cette fonction à la fin de toute utilisation de la liaison SPI. Elle permet en effet de dévalider le bus SPI de l'Arduino tout en laissant l'état des entrées/sorties qu'il utilise inchangé. Elle s'utilise tout simplement sous la forme :

```
SPI.end() ;
```

6.4.3 Exemple d'utilisation d'une liaison SPI

Même si l'utilisation d'une liaison SPI reste relativement simple, un petit exemple n'est pas inutile et ce d'autant que nous vous l'avions promis au § 4.2.6. En effet, nous allons utiliser maintenant un circuit d'interface capable de gérer jusqu'à huit afficheurs 7 segments via une simple liaison SPI, ce qui permet d'économiser de façon drastique les précieuses lignes d'entrées/sorties de l'Arduino puisqu'il ne faudra donc que trois fils pour commander ces afficheurs.

Le circuit que nous avons retenu est très connu puisqu'il s'agit du MAX 7219. Il gère huit afficheurs sept segments en mode multiplexé au travers de son interface série trois fils, parfaitement compatible d'une liaison SPI. Le schéma utilisé pour la réalisation de cet exemple vous est présenté figure 6.15.

© Dunod – La photocopie non autorisée est un délit.

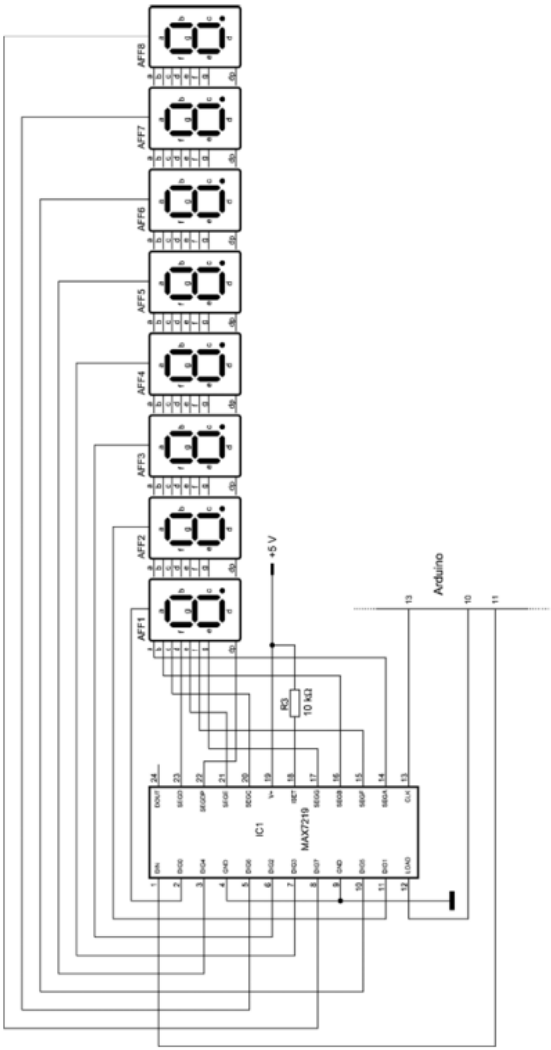

Figure 6.15 – Utilisation d'un MAX 7219 pour gérer jusqu'à huit afficheurs sept segments via une liaison SPI.

Il émane directement de la fiche technique du MAX 7219 que vous trouverez bien évidemment dans les compléments en ligne de cet ouvrage.

La lecture de la fiche technique de ce circuit nous apprend qu'il contient un certain nombre de registres internes définissant son mode de fonctionnement et les données à afficher. Le contenu de ces registres doit lui être transmis sous forme d'un mot de 16 bits (c'est-à-dire encore de deux octets consécutifs) codé comme indiqué figure 6.16. En fait, chaque mot de 16 bits contient sur les 8 bits de poids forts l'adresse du registre destinataire et sur les 8 bits de poids faibles la donnée à écrire dans le registre ainsi sélectionné. En outre, on constate que ces informations ne sont prises en compte que si la ligne LOAD du MAX 7219, préalablement au niveau bas, passe au niveau haut.

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	Adresse Registre				Données							

Figure 6.16 – Contenu des deux octets de dialogue attendus par le MAX 7219.

Dès lors, l'écriture d'un programme de gestion d'un tel circuit est fort simple et nous vous proposons pour cela à titre d'exemple le listing suivant, qui réalise un simple compteur sur trois chiffres avec un MAX 7219.

Listing 6.4 – Exemple de gestion d'un contrôleur d'affichage MAX 7219 au moyen d'une liaison SPI.

```
#include <SPI.h>
#define sspin 10
int chiffre1, chiffre2, chiffre3, compteur ;

void setup()
{
  pinMode(sspin, OUTPUT) ;
  SPI.begin() ;
  // Initialisation du MAX 7219
  ecrimax (0x09, 0xFF); // Décodage en mode BCD
  ecrimax (0x0A, 0x0F); // Intensité affichage maximum
  ecrimax (0x0B, 0x02); // Affichage de 3 chiffres
  ecrimax (0x0C, 0x01); // Validation de l'affichage
  ecrimax (0x00, 0xFF); // Pas de mode test

  // Mise à 0 de tous les chiffres
  for (int i = 0 ; i<8 ; i++)
  {
    ecrimax (i, 0x00) ;
  }
}

void loop()
{
```

© Dunod – La photocopie non autorisée est un délit.

```
  for (compteur = 0 ; compteur < 999; compteur ++)
  {
    chiffre3 = compteur / 100 ; // Extraction des centaines
    chiffre2 = (compteur % 100) / 10 ; // Extraction des dizaines
    chiffre1 = compteur - (100 *chiffre3) - (10 * chiffre2) ;
    ecrimax (0x01,chiffre3) ; // Affichage des centaines
    ecrimax (0x02,chiffre2) ; // Affichage des dizaines
    ecrimax (0x03,chiffre1) ; // Affichage des unités
    delay(500);
  }
}

// Fonction d'écriture dans le MAX 7219
// Ecrit un mot de 16 bits constitué
// de deux mots de 8 bits successifs
// contenant adresse et donnée
void ecrimax(int adresse, int donnee)
{
  digitalWrite(sspin,LOW) ;  // LOAD au niveau bas
  SPI.transfer(adresse) ;    // Envoi du 1er mot de 8 bits
  SPI.transfer(donnee) ;     // Envoi du 2eme mot de 8 bits
  digitalWrite(sspin,HIGH) ; // LOAD au niveau haut
}
```

L'examen du chronogramme de dialogue du MAX 7219 montre que les valeurs affectées par défaut à l'interface SPI de l'Arduino au moyen de l'initialisation en mode standard conviennent. Le programme principal commence donc par un simple `SPI.begin`.

Vient ensuite l'initialisation des registres du MAX 7219 compte tenu du mode de fonctionnement que nous avons choisi. Comme c'était le cas pour la gestion d'un circuit à interface I2C, il faut ici aussi lire avec soin la fiche technique du circuit afin de voir ce qu'il faut écrire dans ses divers registres pour paramétrer telle ou telle fonction. Le listing étant commenté à ce niveau, nous vous laissons le soin de vérifier que les valeurs que nous avons choisies correspondent bien aux fonctions décrites en commentaires.

L'écriture dans le MAX 7219 fait appel à la fonction `ecrimax`, que nous avons créée pour cela, et que l'on trouve en fin de listing. Cette fonction admet deux paramètres en entrée, l'adresse du registre et la donnée à y placer. Elle fait ensuite passer au niveau bas la ligne /SS c'est-à-dire encore la sortie 10 de l'Arduino, que nous avons reliée à la ligne LOAD du MAX 7219 (voir figure 6.15 si nécessaire), avant d'envoyer l'un après l'autre les deux octets d'adresse et de données. La ligne /SS est ensuite remise au niveau haut puisque le MAX 7219 prend en compte les données reçues sur un front montant de LOAD.

Le programme principal quant à lui n'est autre que le compteur qui adopte une structure un peu particulière compte tenu du mode de gestion de l'affichage du MAX 7219. En effet, comme nous voulons faire un compteur à trois chiffres, qui va donc pouvoir aller de 000 à 999, il nous faut faire appel à trois variables différentes appelées `chiffre1`, `chiffre2` et `chiffre3` qui vont représenter chacune un

chiffre du compteur. Le MAX 7219 impose en effet qu'on lui transmette la donnée à afficher chiffre par chiffre. Il est donc impossible de réaliser un comptage avec une seule variable évoluant de 0 à 999.

L'écriture de chaque chiffre fait évidemment appel à la fonction `ecrimax` puisqu'elle ne diffère pas de l'écriture des paramètres d'initialisation vue ci-avant. Seuls les registres destinataires sont différents.

Cet exemple, parfaitement fonctionnel, montre bien la simplicité d'utilisation de la liaison SPI au moyen de la bibliothèque spécialisée proposée par le compilateur. De plus, il vous présente une solution d'affichage multiplexée ainsi que nous vous l'avions annoncé au § 4.2.6.

6.5 Le bus « un fil » de Dallas (Maxim)

Ce bus, créé il y a déjà quelques années par Dallas (qui s'appelle aujourd'hui Maxim suite à son absorption par cette société), permet de relier entre eux tous les circuits disposant d'une interface compatible. Sa particularité est de n'utiliser qu'un fil « actif » et une masse servant de référence de potentiel bien sûr. Ce fil actif véhicule les données dans les deux sens, puisque chaque composant du bus peut être émetteur ou récepteur à tout instant, mais il permet aussi d'alimenter la majorité d'entre eux sous réserve qu'ils ne soient pas trop gourmands en énergie ; c'est ce que l'on appelle alors le mode d'alimentation « parasite ».

Le nombre de circuits disponibles pour ce bus un fil croit sans cesse et les fonctions offertes sont des plus diverses : numéros de série inviolables, mémoires EEPROM, capteurs de température, horloges temps réel, etc.

Il est donc de plus en plus fréquent aujourd'hui que l'on ait à interfacer des circuits dotés de ce bus à un microcontrôleur et donc à notre Arduino. Ce n'est pas particulièrement difficile mais nécessite de connaître au préalable les grands principes qui régissent ce bus un fil.

6.5.1 Principe du bus un fil

Tout d'abord, cette interface étant bidirectionnelle alors qu'elle n'utilise qu'un fil, il est évident qu'elle doit respecter, au niveau de tous les circuits qui y sont connectés, un schéma un peu particulier présenté figure 6.17.

La partie interne des circuits Dallas (Maxim) ne nous concerne évidemment pas puisque nous ne pouvons agir dessus, par contre elle conditionne une partie du schéma d'utilisation du bus un fil. En effet, comme vous pouvez le constater, la sortie du circuit Dallas (Maxim) à destination de ce bus a lieu au moyen d'un transistor MOS à drain ouvert. Il faut donc impérativement ramener ce bus au positif de l'alimentation par une résistance de tirage de 4,7 à 10 kΩ de valeur typique.

© Dunod – La photocopie non autorisée est un délit.

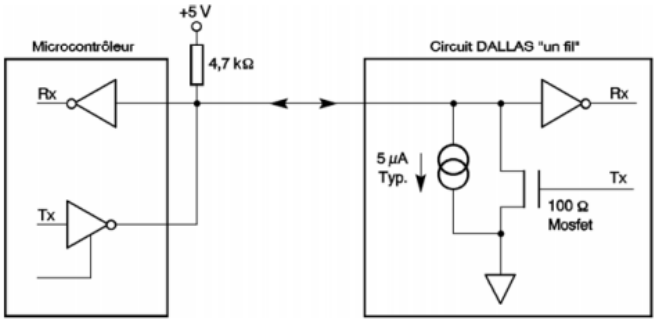

Figure 6.17 – Schéma de l'interface matérielle des circuits à bus « un fil ».

Côté maître du bus, c'est-à-dire côté Arduino dans notre cas, il faut que l'émetteur de données à destination du bus un fil puisse être validé seulement sur commande. Précisons encore que tous les circuits pour bus un fil sont adressables car ils disposent en interne d'un numéro unique codé sur 64 bits, programmé une fois pour toutes lors de leur fabrication. Il est donc possible d'en connecter plusieurs sur le même bus sous le contrôle d'un seul et même Arduino. Ce dernier est alors le maître du bus et les circuits Dallas (Maxim) sont les esclaves selon une terminologie désormais classique.

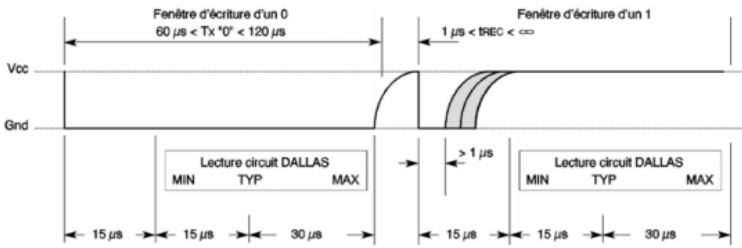

Figure 6.18 – Chronogramme d'émission de données par le maître du bus « un fil ».

Ceci étant vu, la figure 6.18 montre comment le maître du bus émet les deux niveaux logiques possibles à destination des circuits Dallas (Maxim). Pour générer un zéro logique il force tout simplement le bus un fil au niveau logique bas pendant au moins 60 µs, avec une durée maximum qui ne doit pas excéder 120 µs, sachant que le circuit Dallas (Maxim) lit alors le bus entre 15 et 60 µs après sa descente comme schématisé figure 6.18. Pour générer un niveau logique haut par contre, il force le bus un fil au niveau bas (oui, bas, vous avez bien lu, ce n'est pas une coquille) mais pendant une durée comprise entre 1 et 15 µs au maximum. Comme

le circuit Dallas (Maxim) lit toujours le bus au même moment, il voit bien un niveau logique haut puisque celui-ci est alors assuré par la résistance de tirage au positif de l'alimentation.

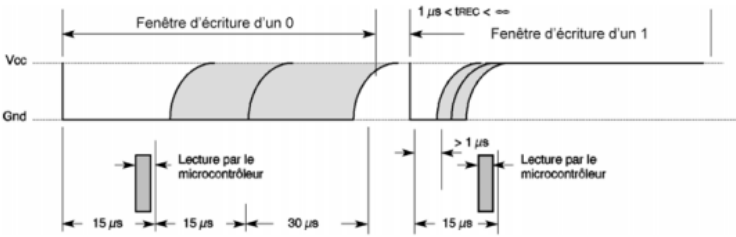

Figure 6.19 – Chronogramme de réception de données par le maître du bus « un fil ».

Pour lire les données émises par un circuit Dallas (Maxim), le procédé est similaire mais repose sur le fait que la sortie des circuits à interface un fil est à drain ouvert. En effet, comme le montre la figure 6.19, le maître du bus force alors ce dernier au niveau bas pendant au moins 15 µs. Si le circuit Dallas (Maxim) veut générer un zéro logique, il maintient alors le bus au niveau bas pendant au moins 15 µs alors que, s'il veut émettre un un logique, il laisse le bus libre et ce dernier remonte donc, sous l'effet de la résistance de tirage au niveau haut, dès que le maître du bus relâche le niveau bas qu'il imposait. Comme le maître du bus doit lire celui-ci à la fin de la fenêtre de 15 µs comme schématisé figure 6.19, il lit bien alors le niveau logique voulu par le circuit Dallas (Maxim).

Pour pouvoir utiliser avec succès les circuits Dallas (Maxim), ce protocole doit évidemment être complété par d'autres informations constituées par les ordres reconnus par tel ou tel circuit ainsi que par le codage des données qu'il fournit en réponse. Ces informations dépendent bien évidemment du circuit utilisé et doivent être extraites de sa fiche technique au cas par cas, exactement comme nous l'avons fait ci-avant pour les circuits à interface I2C ou SPI.

Pour ce qui est de l'alimentation des circuits pour bus un fil ; les moins gourmands d'entre eux peuvent la prélever sur le bus lui-même en exploitant les nombreux instants où il se trouve au niveau logique haut. Il ne faut pas espérer ainsi extraire plus de quelques milli-ampères (mA), surtout si les circuits connectés au bus sont nombreux, mais cela suffit à nombre d'entre eux dont la consommation se chiffre seulement en dizaines ou centaines de micro-ampères.

6.5.2 La bibliothèque OneWire de Jim Studt

Sans être particulièrement compliquée, la programmation du bus un fil nécessite le respect de chronogrammes précis comme nous l'avons vu précédemment et, même si l'on trouve sur Internet quelques exemples de programmes destinés à l'Arduino

© Dunod – La photocopie non autorisée est un délit.

assurant directement la génération de ces signaux, il vaut mieux faire appel pour cela à une bibliothèque prévue à cet effet.

Celle-ci ne faisant pas partie de l'environnement de développement standard au moment où ces lignes sont écrites, il faut faire appel à une bibliothèque tierce dont la plus célèbre est sans conteste celle proposée initialement par Jim Studt, améliorée ensuite par Paul Stoffregen, que vous pourrez télécharger à cette adresse *www.pjrc.com/teensy/arduino_libraries/OneWire.zip* mais que vous trouverez également dans les compléments en ligne au cas où ce lien ne serait plus valide lorsque vous lirez ces lignes. Cette bibliothèque, baptisée tout naturellement `OneWire.h`, que vous prendrez soin d'installer dans le sous-répertoire `librairies` du répertoire d'installation de l'environnement de développement, comme nous l'avons déjà fait à diverses reprises, dispose des fonctions suivantes.

■ OneWire()

Cette fonction doit être appelée avant tout utilisation de la bibliothèque afin de créer une variable (un objet) de type `OneWire`. Cet appel permet également de préciser quelle ligne d'entrée/sortie de l'Arduino sera utilisée pour connecter le bus un fil. Sa syntaxe est en effet la suivante :

```
OneWire UnFil (numéro) ;
```

où `numéro` précise le numéro de la ligne d'entrée/sortie utilisée. L'objet créé dans cet exemple s'appelle `UnFil` (c'est original !) et c'est donc ce nom qui sera utilisé dans tous les exemples de syntaxe suivants.

■ search()

Cette fonction cherche le circuit suivant présent sur le bus un fil, place son adresse dans le tableau qu'elle utilise en paramètre et retourne une valeur binaire `true` (vrai). Si aucun autre circuit n'est trouvé, la fonction retourne une valeur binaire `false` (faux). Sa syntaxe est la suivante :

```
UnFil.search(tableau) ;
```

où `tableau` est une variable indicée à une dimension de huit octets.

■ reset_search()

Cette fonction initie une nouvelle recherche des circuits présents sur le bus un fil. Elle s'utilise tout simplement de la façon suivante :

```
UnFil.reset_search() ;
```

■ reset()

Comme son nom l'indique, cette fonction effectue un reset du bus un fil. Elle est généralement appelée avant toute communication avec un circuit présent sur ce bus. Sa syntaxe est tout simplement :

```
UnFil.reset() ;
```

■ select()

Cette fonction permet de sélectionner un circuit connecté sur le bus un fil au moyen de son adresse, que celle-ci ait été déterminée au préalable par `search` ou qu'elle soit déjà connue par ailleurs par le programme. Une fois cette fonction utilisée, tous les échanges suivants ont lieu avec le circuit sélectionné jusqu'à ce qu'un nouvel appel de cette fonction fasse changer de circuit. Sa syntaxe est la suivante :

```
UnFil.select(tableau) ;
```

où `tableau` est une variable indicée à une dimension de huit octets contenant l'adresse du circuit à sélectionner.

■ skip()

Cette fonction est très utilisée dans le cas très fréquent où un seul circuit est connecté sur le bus un fil. Elle permet en effet de passer outre la recherche des circuits et de leurs adresses et permet donc de communiquer avec le seul circuit présent sur le bus. Elle s'utilise tout simplement sous la forme :

```
UnFil.skip() ;
```

■ write()

Même si vous n'êtes pas anglophone, vous aurez compris que cette fonction sert à écrire une donnée sur le bus ; donnée qui est donc envoyée au circuit préalablement sélectionné. Sa syntaxe est la suivante :

```
UnFil.write(donnée) ;
```

où `donnée` est une variable de type `byte` à envoyer sur le bus un fil.
Elle peut aussi être utilisée sous la forme :

```
UnFil.write(donnée, 1) ;
```

ce qui a pour effet de maintenir le circuit auquel est envoyée la donnée alimentée par l'intermédiaire du bus un fil, si bien sûr ce dernier ne dispose pas d'une alimentation indépendante.

■ read()

Ici aussi pas besoin d'être très versé sur la langue anglaise pour comprendre qu'il s'agit d'une fonction de lecture de la donnée envoyée sur le bus par le circuit préalablement sélectionné. La syntaxe est tout simplement :

```
donnée = UnFil.read() ;
```

■ crc8()

Cette fonction, d'usage moins courant lorsque les conditions d'utilisation du bus un fil ne sont pas critiques, calcule un CRC (un caractère de contrôle particulier si vous préférez) sur une suite de données.

© Dunod – La photocopie non autorisée est un délit.

Il faut savoir en effet que tous les circuits pour bus un fil font suivre les données qu'ils transmettent d'un CRC calculé sur ces dernières. Il suffit donc de le calculer à nouveau sur les données effectivement reçues, puis de le comparer avec celui envoyé par le circuit pour être sûr, on non, de l'intégrité de ces dernières. Toutefois, si le bus un fil n'est pas soumis à des parasites violents ou n'est pas de longueur excessive, le risque d'erreur est faible et la vérification du CRC n'est pas indispensable. Sa syntaxe est la suivante :

```
CRC = UnFil.crc8(tableau, nombre) ;
```

où `tableau` est une variable indicée à une dimension contenant les données sur lesquelles doit être calculé le CRC, `nombre` le nombre de données à utiliser pour ce calcul et `CRC` la variable contenant le résultat du calcul du CRC.

6.5.3 Exemple d'utilisation du bus un fil

Parmi tous les circuits disponibles pour le bus un fil, la famille des DS18x20 de Dallas (Maxim) est certainement la plus utilisée. Ces circuits sont en effet des capteurs de température de précision, dont la mise en œuvre et la possible alimentation via le bus un fil sont en partie responsable du succès.

Il existe actuellement trois versions de DS18x20 qui se distinguent par la lettre centrale de leur référence. Nous avons choisi à titre d'exemple le DS18B20 qui offre une précision de +/– 0,5 °C sur la plage qui va de – 10 °C à + 85 °C et qui peut mesurer de – 55 °C à + 125 °C avec une précision un peu plus réduite dans ce cas. La résolution de mesure peut être programmée quant à elle sur 9 ou 12 bits par le biais d'un registre interne et le circuit dispose en outre d'un registre de définition d'une température d'alarme.

La figure 6.20 présente les deux modes de connexion possible de ce capteur à l'Arduino, selon que l'on souhaite utiliser l'alimentation via le bus un fil ou non. Dans le premier cas la liaison Arduino – capteur ne nécessite alors que deux fils alors qu'il en faut trois dans l'autre cas.

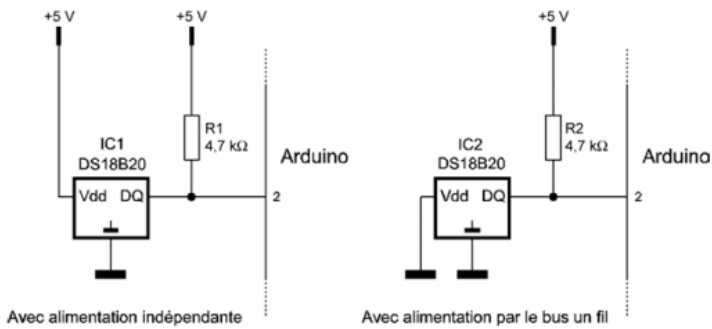

Figure 6.20 – Utilisation d'un capteur de température DS18B20.

Comme nous l'avons déjà dit pour le bus I2C et pour la liaison SPI, la première tâche à accomplir, avant d'écrire le programme de gestion d'un tel circuit, consiste à lire sa fiche technique afin d'identifier les registres internes et les informations que l'on peut y lire ou y écrire. Vous trouverez bien évidemment celle-ci dans les compléments en ligne de cet ouvrage et nous vous laissons le soin de la découvrir avant d'étudier le programme que nous vous proposons.

Listing 6.5 – Exemple de gestion d'un capteur de température DS18B20 sur bus un fil.

```
#include <OneWire.h>  // Bibliothèque OneWire

const int sortieUnFil = 2 ;  // Bus un fil sur 2
const int Mesure = 0x44 ;    // Lancement mesure
const int Lecture = 0xBE ;   // Lecture du capteur

byte donnees[12];      // Tableau pour lecture complète du DS18B0

int tempbrute=0;             // Température brute
float tempreelle=0.0;        // Température réelle

OneWire UnFil(sortieUnFil);  // Crée un objet One Wire
                             // sur la broche voulue

void setup()
{
Serial.begin(9600);          // Liaison série à 9600 bits par
seconde
}

void loop()
{
  UnFil.reset();             // Initialise le bus un fil
  UnFil.skip();              // Un seul capteur donc pas d'adresse
  UnFil.write(Mesure,1);     // Commande de mesure
  delay(1000);
  UnFil.reset();              // Initialise le bus un fil
  UnFil.skip();               // Un seul capteur donc pas d'adresse
  UnFil.write(Lecture,1);     // Lecture des 9 registres du capteur
for ( int i = 0; i < 9; i++)
  {
    donnees[i] = UnFil.read();
  }
Serial.print(«Temperature poids faibles = «), Serial.
println(donnees[0],BIN);
Serial.print(«Temperature poids forts = «), Serial.
println(donnees[1],BIN);
Serial.print(«Alarme haute = «), Serial.println(donnees[2],BIN);
Serial.print(«Alarme basse = «), Serial.println(donnees[3],BIN);
Serial.print(«Configuration = «), Serial.println(donnees[4],BIN);
Serial.print(«Reserve = «), Serial.println(donnees[5],BIN);
Serial.print(«Reserve = «), Serial.println(donnees[6],BIN);
Serial.print(«Reserve = «), Serial.println(donnees[7],BIN);
```

© Dunod – La photocopie non autorisée est un délit.

```
Serial.print(«CRC = «), Serial.println(donnees[8],BIN);
Serial.println(“”);

donnees[1] = donnees[1] & B10000111; // Mise à 0 des bits de signe
tempbrute = donnees[1];              // Récupération poids forts
tempbrute = tempbrute<<8;            // Décalage poids forts
tempbrute = tempbrute+donnees[0];    // Concaténation
                                     // avec poids faibles
tempreelle = float(tempbrute)*6.25;  // Conversion compte tenu de
tempreelle = tempreelle/100.0;       // la résolution du capteur

Serial.print (“Temperature = ”);
Serial.print (tempreelle,2);
Serial.println (” Degres C. ”);
Serial.println(“”);
delay(5000);
}
```

La relative longueur de ce programme n'est pas due au fait que l'utilisation du bus un fil est complexe, bien au contraire, car l'essentiel des instructions ne sert qu'à afficher sur le moniteur de l'environnement de développement les informations délivrées par le capteur. La partie consacrée à la gestion du bus un fil reste en effet très simple grâce à la bibliothèque `OneWire` appelée au début du programme.

On constate en effet que, pour écrire un octet dans le capteur, il suffit de réaliser dans l'ordre l'enchaînement des fonctions `reset`, `skip` (puisqu'il n'y a qu'un capteur et donc pas de détection d'adresse) puis `write`. Ces opérations sont réalisées deux fois de suite : pour ordonner une mesure tout d'abord (code 0x44) puis pour provoquer la lecture des registres internes (code 0xBE).

Ces registres sont lus au moyen d'une boucle qui répète la fonction `read` autant de fois que nécessaire. Tout le reste ne sert ensuite qu'à afficher ce qui a ainsi été lu et, au final, à la conversion de la température brute lue dans le capteur en température réelle sachant que sa résolution par défaut est de 0,0625 °C.

Comme vous pouvez le constater, grâce à la bibliothèque `OneWire`, la gestion d'un circuit à interface un fil est donc d'une grande simplicité.

Ajoutons à propos du DS18x20, qui est décidément très utilisé avec l'Arduino, qu'il existe même une bibliothèque spécifique permettant de lire les informations de température qu'il génère avec encore plus de facilité que la bibliothèque `OneWire` que nous venons de présenter. Si elle vous intéresse, vous la trouverez à cette adresse *http://milesburton.com/wiki/index.php?title=Dallas_Temperature_Control_Library* et également dans les compléments en ligne de cet ouvrage au cas où ce lien ne serait plus valide lorsque vous lirez ces lignes.

7 • POUR ALLER PLUS LOIN

Arrivé à ce stade de l'ouvrage vous devez être à même de concevoir sans difficulté la majorité des applications que l'on peut envisager avec un Arduino puisque, tout au long des chapitres qui précèdent, nous avons vu comment utiliser les principaux « périphériques » qu'il est possible de lui connecter : poussoirs, claviers, afficheurs, relais, moteurs, capteurs, circuits à interface série, I2C, SPI ou bien encore bus « un fil », etc.

Il suffit en effet pour cela d'utiliser nos exemples, que ce soient les schémas ou les programmes, comme des briques de Lego© qu'il ne reste plus qu'à assembler et à adapter pour parvenir à vos fins.

Il est cependant bien évident que nous n'avons pas pu traiter toutes les situations possibles tant les usages de l'Arduino sont aujourd'hui variés. Ainsi, nous n'avons pas vu comment se connecter à Internet, comment utiliser un GPS ou bien encore une carte mémoire SD par exemple.

De même, nous avons donné tout au long de cet ouvrage les informations pour utiliser les Arduino Uno et Mega 2560 mais nous ne nous sommes pas intéressés aux « autres », que ce soient de véritables Arduino ou des « Arduino like ».

Nous vous proposons donc, dans ce chapitre, de voir comment « aller plus loin » en découvrant comment travailler avec des shields spécifiques et/ou complexes, ou bien encore en utilisant tous ces « autres » Arduino aujourd'hui disponibles sur le marché.

7.1 Les shields évolués

Les shields que nous vous avons présentés au chapitre III, ou auxquels nous avons fait appel directement ou indirectement dans les chapitres suivants, sont tous des shields relativement simples au plan des circuits qu'ils utilisent et des logiciels qui sont nécessaires pour leur mise en œuvre. Cependant, si vous avez déjà réalisé quelques recherches sur Internet concernant l'Arduino, vous avez dû constater qu'il existait quelques shields plus évolués. On peut citer à titre d'exemple les shields embarquant un GPS, ceux disposant d'une interface pour carte mémoire SD ou bien encore ceux d'interface Ethernet.

© Dunod – La photocopie non autorisée est un délit.

7.1.1 Programmation des shields évolués

Sans être difficile outre mesure, la programmation de ces shields évolués nécessite cependant une étude préalable relativement longue car il est souvent nécessaire de posséder un minimum de connaissance de la fonction qu'ils accomplissent avant de pouvoir les gérer. Il est ainsi inconcevable de vouloir récupérer, dans une application, les informations délivrées par un GPS sans avoir étudié au préalable la structure des trames NMEA. Ce sont en effet ces trames qui contiennent les informations intéressantes et qui sont fournies par tout module GPS qui se respecte.

Heureusement, la majorité des fournisseurs qui mettent sur le marché des shields évolués pour l'Arduino proposent en même temps, au minimum une bibliothèque spécifique, et bien souvent des exemples de mise en œuvre de cette dernière.

Avant de choisir un shield évolué, il est donc vivement conseillé de vérifier si la bibliothèque qui va avec existe car, dans le cas contraire et sauf à maîtriser parfaitement ce que supporte le shield, vous risquez de vous retrouver avec de sérieuses difficultés de programmation.

À ce propos, nous tenons à vous mettre en garde à propos d'un certain nombre de shields, disponibles notamment sur eBay, qui sont proposés sans aucune documentation ni bibliothèque associée. À notre avis ces produits sont à éviter autant que faire se peut pour les raisons que nous venons d'exposer. La maigre économie éventuellement réalisée lors de leur achat est bien vite compensée par la perte de temps qui risque de résulter de leur difficulté de programmation.

En présence d'un shield évolué, de sa bibliothèque associée et des éventuels exemples qui vont avec, il ne faut alors pas hésiter à s'inspirer très fortement d'un de ces derniers pour développer votre application. Les exemples fournis avec un shield sont effet là dans ce but et sont généralement relativement bien écrits et commentés (en anglais hélas diront certains, mais l'anglais technique est assez facile à lire !).

En présence d'un problème de programmation d'un de ces shields, n'oubliez pas non plus que, comme on le lit souvent sur les forums, « Google est votre ami ». Internet regorge en effet aujourd'hui d'applications destinées à l'Arduino, et il serait bien rare que vous ne trouviez pas, sinon la solution directe de votre problème, au moins une solution approchante que vous pourrez ensuite adapter.

Enfin bien sûr nous ne saurions trop vous recommander la lecture de notre autre ouvrage consacré à l'Arduino, intitulé *Arduino, Applications avancées* (même auteur, même éditeur), qui aborde justement l'utilisation matérielle et la programmation de nombreux shields évolués. Y sont ainsi abordés, par exemple, les claviers tactiles, les écrans graphiques, les GPS, les modules XBee mais aussi le traitement des sons et des images.

7.1.2 Reliez votre Arduino à Internet

Afin de mettre en pratique ce que nous venons d'écrire, nous vous proposons de relier votre application à base d'Arduino à Internet afin, par exemple, que vous puissiez consulter à distance la température qu'il fait chez vous quand vous n'y

êtes pas. Pour cela, nous allons faire appel à un shield très évolué puisqu'il s'agit de l'Arduino Ethernet Shield.

Comme il existe aujourd'hui plusieurs versions d'un tel shield, nous avons fait appel à celle qui est fabriquée en Italie et proposée par les concepteurs même de l'Arduino. Ce shield est présenté sur le site officiel consacré à l'Arduino sur la page dont voici l'adresse : *http://arduino.cc/en/Main/ArduinoEthernetShield* et il est disponible en France chez Lextronic (*www.lextronic.fr*).

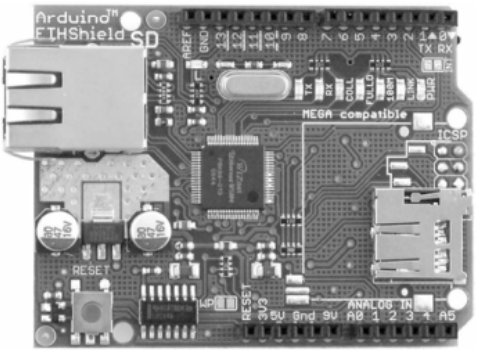

Figure 7.1 – L'Arduino Ethernet Shield utilisé dans cet exemple (doc. www.arduino.cc).

Il est évidemment proposé avec une bibliothèque très complète, installée d'origine avec l'environnement de développement de l'Arduino, ainsi qu'avec de nombreux exemples de programmes.

Pour bien l'utiliser, et comme nous l'évoquions ci-dessus à propos d'un shield GPS pris en exemple, il est évidemment souhaitable de posséder quelques notions relatives à Ethernet car, dès que vous allez commencer à le programmer, la documentation de la bibliothèque associée et les exemples fournis vont vous parler d'adresses IP et MAC, de pile IP, de TCP, d'UDP et de bien d'autres sigles barbares. Il est cependant possible de démarrer « doucement », comme nous l'avons expliqué ci-dessus, en partant d'un des exemples fournis avec le shield.

Nous souhaitons en effet pouvoir consulter à distance, c'est-à-dire par notre réseau ou par Internet, l'état d'un ou plusieurs capteurs de température situés dans un local pour connaître la température qu'il y fait pendant notre absence. Pour la mesure de température, nous avons le choix entre les capteurs à bus « un fil » DS 18x20, que nous avons vus au chapitre 6, ou bien les capteurs analogiques TMP36 présentés au chapitre 5. Afin que notre exemple reste simple, nous allons utiliser ces derniers et vous renvoyons donc, si nécessaire, à ce même chapitre 5 pour voir comment les câbler et les programmer.

© Dunod – La photocopie non autorisée est un délit.

Comme nous voulons pouvoir consulter les données acquises par l'Arduino via Internet, notre application doit se comporter comme un serveur Internet c'est-à-dire qu'elle doit fournir ses données suite à des requêtes que nous lui enverrons.

Fort heureusement, il existe un exemple parfaitement adapté à cela parmi ceux fournis avec cette carte. Il s'appelle tout simplement `WebServer` et vous pouvez y accéder directement via le menu **Fichier**, sous-menu **Exemple** de l'Arduino. Nous reproduisons son listing ci-dessous *in extenso*, avec ses commentaires anglais d'origine, afin de vous montrer comment l'adapter à notre besoin.

Listing 7.1 – Le programme de serveur web (WebServer) fourni à titre d'exemple avec l'Arduino Ethernet Shield.

```
/*
  Web  Server

 A simple web server that shows the value of the analog input pins.
 using an Arduino Wiznet Ethernet shield.

 Circuit:
 * Ethernet shield attached to pins 10, 11, 12, 13
 * Analog inputs attached to pins A0 through A5 (optional)

 created 18 Dec 2009
 by David A. Mellis
 modified 4 Sep 2010
 by Tom Igoe

 */

#include <SPI.h>
#include <Ethernet.h>

// Enter a MAC address and IP address for your controller below.
// The IP address will be dependent on your local network:
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192,168,1, 177 };

// Initialize the Ethernet server library
// with the IP address and port you want to use
// (port 80 is default for HTTP):
Server server(80);

void setup()
{
  // start the Ethernet connection and the server:
  Ethernet.begin(mac, ip);
  server.begin();
}

void loop()
{
  // listen for incoming clients
```

```
    Client client = server.available();
    if (client) {
      // an http request ends with a blank line
      boolean currentLineIsBlank = true;
      while (client.connected()) {
        if (client.available()) {
          char c = client.read();
          // if you've gotten to the end of the line (received a
          ➥newline
          // character) and the line is blank, the http request has
          ➥ended,
          // so you can send a reply
          if (c == '\n' && currentLineIsBlank) {
            // send a standard http response header
            client.println(«HTTP/1.1 200 OK»);
            client.println(«Content-Type: text/html»);
            client.println();

            // output the value of each analog input pin
            for (int analogChannel = 0; analogChannel < 6;
            ➥analogChannel++) {
              client.print(«analog input «);
              client.print(analogChannel);
              client.print(« is «);
              client.print(analogRead(analogChannel));
              client.println(«<br />»);
            }
            break;
          }
          if (c == '\n') {
            // you're starting a new line
            currentLineIsBlank = true;
          }
          else if (c != '\r') {
            // you've gotten a character on the current line
            currentLineIsBlank = false;
          }
        }
      }
      // give the web browser time to receive the data
      delay(1);
      // close the connection:
      client.stop();
    }
  }
```

L'examen de la documentation du shield Ethernet nous apprend tout d'abord que, comme celui-ci supporte aussi une interface pour carte SD et que cette dernière partage son interface de type SPI avec le contrôleur Ethernet, il faut invalider l'un ou l'autre en agissant sur les sorties 4 et 10 de l'Arduino. Ainsi, le fait de mettre la sortie 4 au niveau haut invalide l'interface carte SD alors que le fait de mettre 10 au niveau haut invalide l'interface Ethernet. Ceci étant vu, l'exemple de programme fourni se laisse facilement analyser, tout du moins pour ce que nous voulons en faire.

© Dunod – La photocopie non autorisée est un délit.

Il faut tout d'abord commencer par définir l'adresse MAC du shield à l'emplacement prévu à cet effet. Cette adresse est en principe indiquée sur une étiquette collée sous le shield. En son absence, indiquez n'importe quelle adresse MAC (une combinaison de six nombres hexadécimaux à deux chiffres) en veillant toutefois à ce que la valeur choisie ne corresponde pas à celle d'un des éléments se trouvant sur le réseau auquel sera connecté le shield.

Indiquez ensuite l'adresse IP qui sera affectée au shield. Il faut en choisir une qui soit adaptée à la plage d'adresses permise par le réseau auquel sera connecté le shield. Pour les réseaux domestiques, elle est généralement de la forme 192.168.0.xxx ou 192.168.1.xxx alors que pour les réseaux d'entreprise elle peut être quelconque, mais aucune restriction n'est faite par la bibliothèque ou par le shield à ce niveau.

Notez également que, sur les dernières versions de cette bibliothèque Ethernet, la notion de DHCP est désormais supportée. La définition de l'adresse IP de la carte n'est alors plus indispensable pour peu que le réseau auquel elle sera connectée dispose d'un serveur DHCP.

Vient ensuite la définition du port qui va être utilisé pour dialoguer avec la carte. Comme nous allons travailler en http via un navigateur Internet, on peut laisser la valeur 80 qui est proposée par défaut.

Dès cet instant, il n'est plus nécessaire de modifier quoi que ce soit pour que l'application fonctionne et, si vous la compilez et que vous lanciez son exécution, il vous sera possible d'accéder à son adresse IP, soit via votre réseau domestique, soit via Internet si ce dernier a un accès externe. Vous verrez alors s'afficher dans la fenêtre du navigateur utilisé les informations reproduites figure 7.2.

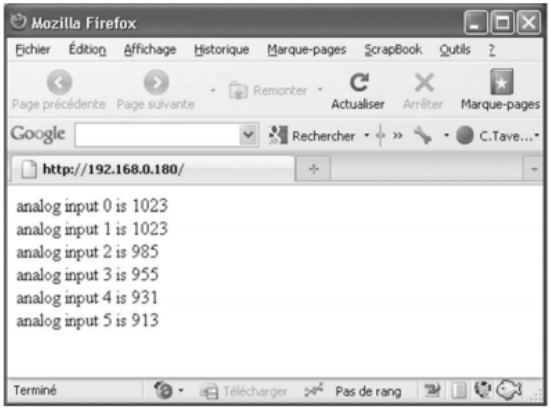

Figure 7.2 – L'affichage par défaut de l'exemple WebServer.

Comme vous pouvez le constater, on peut bien prendre connaissance à distance de l'état des six entrées analogiques de l'Arduino, sans avoir écrit quasiment aucune ligne de programme et… presque sans rien comprendre à Ethernet ni aux réseaux !

Pour adapter ce programme à notre besoin, il suffit juste de modifier la partie du listing reproduit ci-après et de la remplacer par ce que nous voulons faire afficher par le navigateur.

Listing 7.2 – La partie du programme WebServer à adapter à notre besoin.

```
for (int analogChannel = 0; analogChannel < 6; analogChannel++) {
            client.print(«analog input «);
            client.print(analogChannel);
            client.print(« is «);
            client.print(analogRead(analogChannel));
            client.println(«<br />»);
          }
          break;
```

C'est en effet cette seule partie qui gère l'information textuelle envoyée au navigateur lorsque vous accédez au serveur réalisé au moyen du shield Ethernet.

Ainsi par exemple, si nous utilisons pour mesurer la température un TMP36 connecté sur l'entrée analogique A4 de l'Arduino, et que nous ne souhaitons voir que cette seule information lorsque nous interrogeons notre serveur, il suffira de remplacer le listing 7.2 par celui présenté ci-après.

Listing 7.3 – Adaptation du programme WebServer à l'affichage de la température.

```
tension = map(analogRead(capteur),0,1023,0,500);
          temperature = (tension - 50) ;
          client.print(«Temperature : «) ;
          client.print(temperature) ;
          client.println(«<br />»);
          break;
```

La partie mesure et conversion de la température est tout simplement reprise du listing 5.4 vu au chapitre 5, tandis que les données envoyées sous forme textuelle par le programme `WebServer` sont remplacées par celles qui nous conviennent.

Si vous trouvez l'affichage quelque peu tristounet, vous pouvez très bien ajouter autant de `client.print` que vous le désirez afin de formater celui-ci au moyen des règles de la syntaxe du langage html.

Nous en resterons là de cet exemple car notre but était de vous montrer que, même en présence d'un shield complexe, il est possible de l'utiliser et de l'adapter à ses besoins pour peu qu'il dispose d'une bibliothèque appropriée et d'exemples de mise en œuvre suffisamment commentés.

De nombreux exemples détaillés de connexion à un réseau Ethernet, et donc à Internet, sont par ailleurs proposés dans notre ouvrage *Arduino, Applications avancées* évoqué précédemment.

© Dunod – La photocopie non autorisée est un délit.

7.2 Les autres Arduino originaux

Même si les Arduino les plus répandus et les plus utilisés sont encore aujourd'hui ceux que nous avons retenus dans le cadre de cet ouvrage, à savoir les Uno et, pour les applications très gourmandes en entrées/sorties, le Mega 2560 ; la famille comporte de nombreux autres membres que nous vous proposons de découvrir rapidement ci-après.

Compte tenu de leur aspect physique, certains diraient de leur « packaging », ils peuvent être mieux adaptés que d'autres pour des applications ayant, par exemple, des contraintes d'encombrement.

7.2.1 L'Arduino Mini

Comme vous pouvez le constater figure 7.3, l'Arduino Mini se présente sous la forme d'un pseudo-circuit intégré 24 pattes, similaire dans son aspect au célèbre Basic Stamp de Parallax (*www.parallax.com*), et peut de ce fait être directement implanté sur le circuit imprimé d'une application, par soudure ou par enfichage dans un support normalisé pour circuits intégrés de cette taille.

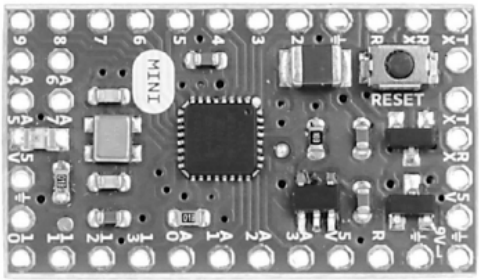

Figure 7.3 – L'Arduino Mini.

Proposée à l'origine avec un ATmega168, ce qui lui conférait des caractéristiques plus limitées que celles de l'Arduino Uno comme le résume le tableau 7.1, les dernières versions de l'Arduino Mini commercialisées à ce jour sont désormais équipées d'un ATmega 328. Ses possibilités sont donc identiques à celles d'un Arduino Uno si ce n'est bien sûr l'absence de possibilité de connexion directe de shields ainsi hélas que, compte tenu de sa taille réduite, l'absence de port USB.

Tableau 7.1 – Principales caractéristiques de l'Arduino Mini.

Paramètre	Mini	Uno
Microcontrôleur	ATmega 168	ATmega 328
Alimentation	5 V	5 V
Tension d'entrée	7 – 9 V	7 – 12 V
Entrées/sorties numériques	14	14
Courant maximum d'E/S	40 mA	40 mA
Sorties PWM	6	6
Entrées analogiques	8	6
Mémoire de programme	16 K	32 K
Mémoire EEPROM	512 octets	1 K
Mémoire vive (RAM)	1 K	2 K
Horloge	16 MHz	16 MHz
USB	Non	Oui
Dimensions	18 x 33 mm	65 x 52 mm

Pour pouvoir le programmer et/ou développer une application avec le Mini, il faut donc acquérir en même temps un petit circuit d'interface, appelée Mini USB Adapter, dont l'aspect est visible figure 7.4. Ce circuit imprimé assure la conversion entre le port USB du PC et les entrées/sorties séries Rx et Tx du Mini ainsi que son alimentation.

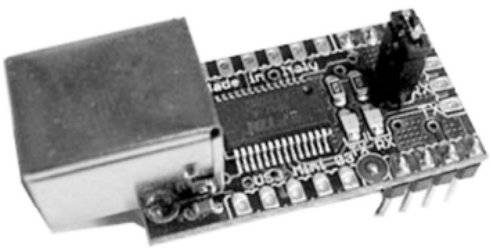

Figure 7.4 – L'adaptateur USB Mini USB Adapter.

© Dunod – La photocopie non autorisée est un délit.

Du fait de l'absence de cette interface de programmation, l'Arduino Mini est moins coûteux que les versions qui en sont équipées et, comme cette interface ne sert qu'en phase de développement de programme, cela destine donc plus le Mini à être utilisé dans des applications finalisées.

7.2.2 L'Arduino Nano

Une rapide analyse pourrait faire croire que l'Arduino Nano est plus intéressant que le Mini et qu'il est amené à le supplanter. Il est en effet presque aussi peu encombrant puisque, comme le montre la figure 7.5, il revêt aussi l'aspect d'un circuit intégré, mais à 30 pattes cette fois-ci.

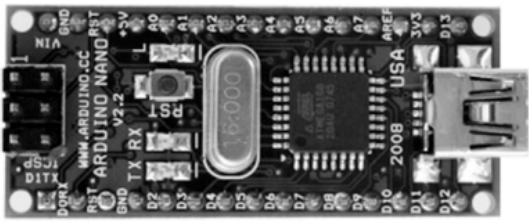

Figure 7.5 – L'Arduino Nano (doc. www.arduino.cc).

Par contre, il intègre d'origine l'interface USB destinée au développement de programmes et à sa programmation. Compte tenu de l'encombrement du Nano, cette interface est munie d'une prise mini USB mais cela ne change bien sûr rien à son principe.

Les versions 2.x de l'Arduino Nano sont équipées d'un ATmega 168 tandis que les versions 3.x embarquent un ATmega 328. Leurs caractéristiques sont par ailleurs synthétisées dans le tableau 7.2 et, comme vous pouvez le constater, sont identiques à celles de l'Uno.

En ce qui nous concerne, nous ne voyons pas très bien l'intérêt de l'Arduino Nano. En effet, ce n'est jamais qu'un Arduino Uno, coûtant près du double en raison de sa petite taille, et surtout étant incapable de se raccorder directement aux très nombreux shields disponibles pour les Arduino « de taille normale ».

Son seul intérêt pourrait être sa possible intégration dans une application finalisée, en raison de son faible encombrement, mais son prix nous semble dissuasif pour ce faire.

Tableau 7.2 – Principales caractéristiques des Arduino Nano.

Paramètre	Nano 2.x	Nano 3.x	Uno
Microcontrôleur	ATmega 168	ATmega 328	ATmega 328
Alimentation	5 V	5 V	5 V
Tension d'entrée	7-12 V	7-12 V	7-12 V
Entrées/sorties numériques	14	14	14
Courant maximum d'E/S	40 mA	40 mA	40 mA
Sorties PWM	6	6	6
Entrées analogiques	8	8	8
Mémoire de programme	16 K	32 K	32 K
Mémoire EEPROM	512 octets	1 K	1 K
Mémoire vive (RAM)	1 K	2 K	2 K
Horloge	16 MHz	16 MHz	16 MHz
USB	Oui Mini	Oui Mini	Oui
Dimensions	18 x 43 mm	18 x 43 mm	65 x 52 mm

7.2.3 L'Arduino Fio

Plus original que les précédents et à peine plus encombrant, l'Arduino Fio est destiné tout à la fois aux applications alimentées par batteries et communicantes sans fil. Il dispose en effet, outre les principaux éléments qui ont fait le succès de l'Arduino, d'un connecteur pour batterie Lithium Polymère, avec un circuit de charge via le port USB, ainsi que d'un connecteur pour module sans fil de type Xbee.

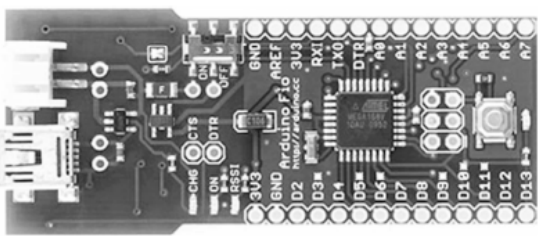

Figure 7.6 – L'Arduino Fio (doc. www.arduino.cc).

© Dunod – La photocopie non autorisée est un délit.

Il est équipé d'un ATmega 328P, fonctionnant sous 3,3 volts, et il convient donc d'en tenir compte lors de la réalisation de circuits d'interface. En effet, un certain nombre de shields destinés aux Arduino Uno ne fonctionnent pas correctement sous cette tension. Par ailleurs, par soucis d'économie d'énergie, sa fréquence d'horloge a été réduite à 8 MHz. Les programmes s'exécutent donc deux fois moins vite sur un Fio que sur un Uno.

Hormis ces deux différences majeures, les autres caractéristiques sont quasiment identiques à celles de l'Arduino Uno comme indiqué tableau 7.3.

Tableau 7.3 – Principales caractéristiques de l'Arduino Fio.

Paramètre	Fio	Uno
Microcontrôleur	ATmega 328P	ATmega 328
Alimentation	3,3 V	5 V
Tension d'entrée	3,35-12 V	7-12 V
Entrées/sorties numériques	14	14
Courant maximum d'E/S	40 mA	40 mA
Sorties PWM	6	6
Entrées analogiques	8	6
Mémoire de programme	32 K	32 K
Mémoire EEPROM	1 K	1 K
Mémoire vive (RAM)	2 K	2 K
Horloge	8 MHz	16 MHz
USB	Oui Mini	Oui
Spécificité	Chargeur de batterie sur USB	–
Dimensions	28 x 66 mm	65 x 52 mm

7.2.4 L'Arduino Pro

Destiné à être monté dans des applications semi-permanentes selon son promoteur, la société Sparkfun (*www.sparkfun.com*), l'Arduino Pro, ou plutôt les Arduino Pro car ils sont plusieurs, adoptent l'aspect visible figure 7.7.

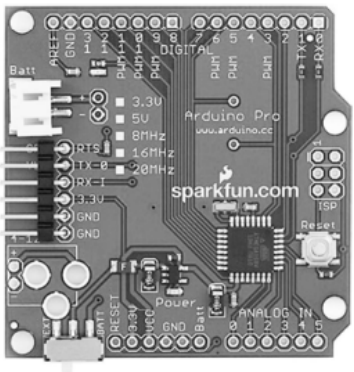

Figure 7.7 – L'Arduino Pro (doc. Sparkfun).

Comme vous pouvez le constater, les connecteurs montés à demeure sur les Uno sont absents, mais des trous pastillés réalisés dans le circuit imprimé se trouvent aux mêmes endroits que sur ces derniers. Cela signifie que, si nécessaire, il reste possible d'y souder les classiques connecteurs au pas de 2,54 mm permettant l'enfichage direct de shields standards.

L'alimentation peut avoir lieu à partir d'une batterie, pour laquelle un connecteur est prévu, via un jack et une alimentation externe à condition de souder un tel jack dans l'emplacement prévu à cet effet, ou via le port USB de liaison au PC.

Tout comme le Mini, le Pro ne dispose pas d'origine de cette interface USB et il faut donc recourir, pour le programmer, au module Mini USB Adapter présenté au § 7.2.1.

L'Arduino Pro existe à l'heure actuelle en deux versions qui se distinguent par leur tension d'alimentation et leur fréquence d'horloge comme vous pouvez le constater dans le tableau synthétique de leurs caractéristiques.

Tableau 7.4 – Principales caractéristiques des Arduino Pro.

Paramètre	Pro 328/3,3 V	Pro 328/5 V	Uno
Microcontrôleur	ATmega 328	ATmega 328	ATmega 328
Alimentation	3,3 V	5 V	5 V
Tension d'entrée	3,35-12 V	5-12 V	7-12 V
Entrées/sorties numériques	14	14	14
Courant maximum d'E/S	40 mA	40 mA	40 mA

© Dunod – La photocopie non autorisée est un délit.

Paramètre	Pro 328/3,3 V	Pro 328/5 V	Uno
Sorties PWM	6	6	6
Entrées analogiques	6	6	6
Mémoire de programme	32 K	32 K	32 K
Mémoire EEPROM	1 K	1 K	1 K
Mémoire vive (RAM)	2 K	2 K	2 K
Horloge	8 MHz	16 MHz	16 MHz
USB	Non	Non	Oui
Dimensions	53 x 52 mm	53 x 52 mm	65 x 52 mm

7.2.5 L'Arduino Pro Mini

Produit également par Sparkfun (*www.sparkfun.com*), l'Arduino Pro Mini reprend l'aspect physique du Mini comme vous pouvez le constater sur la figure 7.8.

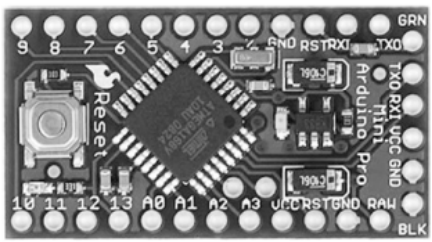

Figure 7.8 – L'Arduino Pro Mini (doc. www.arduino.cc).

Par contre, et contrairement au Mini, le Pro Mini est équipé d'un ATmega 328 et peut fonctionner sous 3,3 volts ou 5 volts selon les versions. Le tableau 7.5 présente donc les caractéristiques qui en résultent.

Tout comme le Mini, le Pro Mini ne dispose pas de l'interface USB de liaison au PC et de programmation et, pour pouvoir le programmer et/ou développer une application, il faut donc acquérir, là aussi, le Mini USB Adapter déjà présenté ci-avant.

Tableau 7.5 – Principales caractéristiques des Arduino Pro Mini.

Paramètre	Pro Mini 328/3,3 V	Pro Mini 328/5 V	Uno
Microcontrôleur	ATmega 328	ATmega 328	ATmega 328
Alimentation	3,3 V	5 V	5 V
Tension d'entrée	3,35-12 V	5-12 V	7-12 V
Entrées/sorties numériques	14	14	14
Courant maximum d'E/S	40 mA	40 mA	40 mA
Sorties PWM	6	6	6
Entrées analogiques	6	6	6
Mémoire de programme	32 K	32 K	32 K
Mémoire EEPROM	1 K	1 K	1 K
Mémoire vive (RAM)	2 K	2 K	2 K
Horloge	8 MHz	16 MHz	16 MHz
USB	Non	Non	Oui
Dimensions	18 x 33 mm	18 x 33 mm	65 x 52 mm

7.2.6 L'Arduino LilyPad

Terminons cette présentation des modules Arduino « officiels » par la version certainement la plus originale de toutes, au moins par son aspect, puisqu'il s'agit de l'Arduino LilyPad qui ressemble… à une fleur comme vous pouvez le constater figure 7.9.

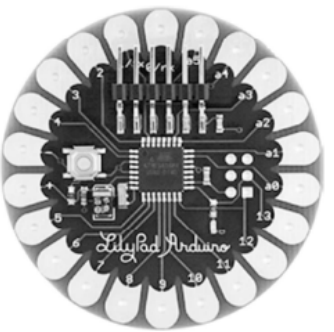

Figure 7.9 – L'Arduino LilyPad (doc. www.arduino.cc).

© Dunod – La photocopie non autorisée est un délit.

Cette version, à l'allure si particulière, a été conçue à l'origine pour être intégrée, par exemple, à des vêtements intelligents ou e-textiles comme les appellent leurs promoteurs. Elle existe actuellement en deux versions dont les caractéristiques sont résumées tableau 7.6. Bien sûr, compte tenu de son aspect physique, le LilyPad ne comporte pas d'interface ni de connecteur USB et il faut donc faire appel au désormais célèbre Mini USB Adapter pour le programmer.

Tableau 7.6 – Principales caractéristiques des Arduino LilyPad.

Paramètre	LilyPad 168V	LilyPad 328V	Uno
Microcontrôleur	ATmega 168V	ATmega 328V	ATmega 328
Alimentation	2,7-5,5 V	2,7-5,5 V	5 V
Tension d'entrée	2,7-5,5 V	2,7-5,5 V	7-12 V
Entrées/sorties numériques	14	14	14
Courant maximum d'E/S	40 mA	40 mA	40 mA
Sorties PWM	6	6	6
Entrées analogiques	6	6	6
Mémoire de programme	16 K	32 K	32 K
Mémoire EEPROM	512 octets	1 K	1 K
Mémoire vive (RAM)	1 K	2 K	2 K
Horloge	8 MHz	8 MHz	16 MHz
USB	Non	Non	Oui
Dimensions	Diamètre 50 mm	Diamètre 50 mm	65 x 52 mm

Les entrées/sorties quant à elles sont disponibles sous forme de larges pastilles, disposées sur le pourtour de la carte, sur lesquelles il est possible de souder des fils de connexion, de se raccorder avec de la peinture ou résine conductrice, ou bien encore du circuit imprimé souple.

7.3 Les « Arduino like »

Le succès de l'Arduino aidant, un certain nombre de cartes inspirées de son concept ont vu le jour sur le marché. Certaines ne sont que des clones, qui ne font que reprendre les schémas des Uno par exemple sans apporter quoi que ce soit de nouveau. Nous n'en parlerons donc pas ici car c'est totalement sans intérêt.

Par contre, un certain nombre de développeurs ont cherché à améliorer le concept de l'Arduino ce qui a conduit à la fabrication de modules, souvent directement compatibles avec les Uno, mais présentant des fonctionnalités supplémentaires ou originales.

Dans le cadre de cet ouvrage, et parmi les dizaines de références aujourd'hui disponibles, nous avons estimé que trois d'entre elles pouvaient retenir votre attention en raison de leurs particularités.

7.3.1 Le Eleven, anciennement TwentyTen de Freetronics

Cette version de l'Arduino Uno, puisqu'elle en est 100 % compatible, est à notre avis très intéressante pour toutes les applications où l'on n'a que très peu de composants d'interface à ajouter à l'Arduino. En effet, comme le montre la figure 7.10, cette carte adopte les mêmes dimensions et dispositions de connecteurs que les Uno mais, grâce à un dessin de circuit imprimé différent, elle offre une zone pastillée vierge de tout composant.

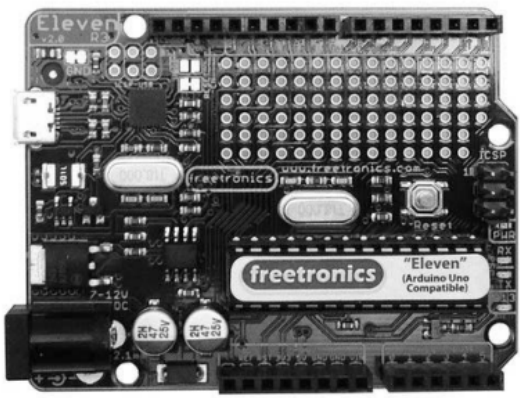

Figure 7.10 – La carte Eleven de Freetronics (doc. Freetronics).

Cette zone pastillée permet ainsi de câbler un certain nombre de circuits d'interfaces élémentaires, une horloge à base de DS 1307 par exemple, conduisant ainsi à la réalisation d'un système très compact ou n'imposant pas le recours à shield prototype qui serait alors surdimensionné.

Un certain nombre d'autres améliorations ont également été apportées aux Arduino Uno avec, par exemple :

– le fait que les LED soient désormais visibles sur le côté de la carte, ce qui évite de les masquer lors de la mise en place d'un shield ;

© Dunod – La photocopie non autorisée est un délit.

– le remplacement du connecteur USB par un modèle mini USB ce qui élimine le risque de court-circuit avec les pistes de la face inférieure du premier shield empilé sur l'Arduino ;
– un meilleur repérage des entrées/sorties, côté composants et côté cuivre du circuit imprimé, facilitant ainsi le câblage ;
– l'absence de perturbation pouvant parfois être causée par la LED connectée à demeure sur la patte 13.

À l'heure actuelle, cette carte n'est semble-t-il disponible que par vente en ligne directement depuis le site de son concepteur, la société Freetronics, d'adresse *www.freetronics.com*.

7.3.2 Le Freeduino

Alors que le projet Arduino est de type « open source » ainsi que nous l'avons mentionné dans l'avant-propos de cet ouvrage, le nom Arduino est une marque déposée et les dessins des circuits imprimés Arduino originaux sont également protégés.
Un projet totalement « open source » a donc vu le jour dans le but de remédier à cet état de fait et propose aujourd'hui le Freeduino ou, plus exactement les Freeduino. Sous cette appellation, et en se rendant sur le site qui leur est consacré (*www.freeduino.org*) on peut librement télécharger tous les fichiers nécessaires à la réalisation de plusieurs versions de circuits compatibles de l'Arduino Uno ou de l'ancien Duemilanove mais utilisant, soit des composants conventionnels, soit des composants CMS aux formats 0603, 0805 ou 1206. La figure 7.11 présente ainsi un exemple de carte Freeduino à base de composants CMS de taille 0603.

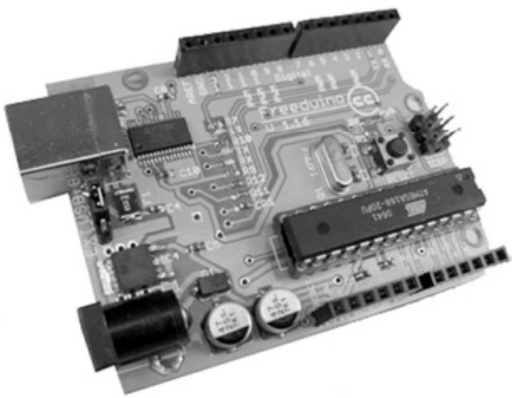

Figure 7.11 – Une carte Freeduino version CMS 0603 (doc. Freeduino.org).

Ces dessins étant totalement libres de droits, divers fabricants proposent « leurs » versions de Freeduino qui cependant restent toutes compatibles des Uno et Duemilanove, parfois avec des petits « plus » tels que, par exemple et sans que ce ne soit exhaustif : connecteur mini USB, interrupteur marche/arrêt, interface série RS 232 à la place du port USB, etc.

7.3.3 Le Boarduino

Ce produit, proposé par Adafruit Industries (*www.adafruit.com*), est particulièrement original et peut rendre service en phase de conception d'une application du fait de son aspect très particulier.

Il s'agit en effet d'un clone parfait de l'Arduino mais d'un format très réduit et surtout adapté aux plaques de câblage rapide standards dans lesquelles il peut s'enficher directement comme on peut le voir sur la figure 7.12.

Il permet ainsi de se passer totalement de shield prototype lorsque l'on souhaite, par exemple, développer une application qui ne peut se satisfaire d'aucun shield standard, ou bien encore si l'on veut mettre au point un schéma d'interface dans le but de réaliser ensuite son propre shield.

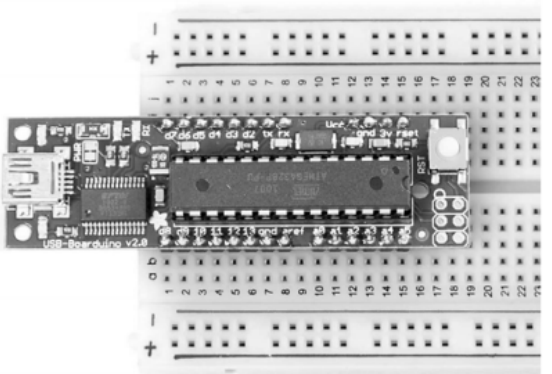

Figure 7.12 – Le Boarduino version USB enfiché sur une plaque de câblage rapide (doc. Adafruit).

Le Boarduino actuel est équipé d'un ATmega 328 fonctionnant à 16 MHz et dispose de toutes les fonctionnalités d'un Arduino Uno. Il existe en deux versions différentes : celle présentée figure 7.12 disposant d'un connecteur mini USB et donc de l'interface avec un PC et l'environnement de développement, qui s'appelle officiellement USB Boarduino, et une version baptisée DC Boarduino où le connecteur USB est remplacé par un jack destiné à recevoir une alimentation externe. Pour le programmer il faut alors recourir à une interface série USB tel le

© Dunod – La photocopie non autorisée est un délit.

Mini USB Adapter que vous commencez à bien connaître ou un de ses équivalents, vendus généralement sous l'appellation FTDI USB – RS 232 Adapter, ou encore FTDI Cable lorsque le circuit d'interface est suffisamment miniaturisé pour être intégré dans la prise USB.

7.4 Les « gros » Arduino

Le succès indéniable de l'Arduino n'a pas incité ses concepteurs à se reposer sur leurs lauriers bien au contraire, et de nombreux nouveaux Arduino ont ou sont en train de voir le jour.

Hormis les variantes « cosmétiques » que nous venons de décrire, ou les mises à jour de l'Arduino Uno ou du Mega 2560, qui ne sont que des ajustements mineurs autour du processeur de départ, on trouve aujourd'hui sur le marché de nouveaux Arduino dont les performances vont en croissant, tant pour ce qui est du nombre d'entrées/sorties disponibles que pour ce qui est des possibilités de programmation.

Dans le cadre de cet ouvrage, il n'est bien sûr pas possible de vous les présenter en détail car un livre entier serait nécessaire pour ce qui est des Arduino Due et Tre ou Galileo dont nous parlerons dans un instant. Nous allons nous arrêter par contre un instant sur un des « successeurs » du Uno qui a pour nom Leonardo.

7.4.1 L'Arduino Leonardo

Bien qu'il ressemble extérieurement à un Arduino Uno, comme on peut le constater figure 7.13, l'Arduino Leonardo recèle quelques particularités qui pourraient bien en faire, dans un avenir relativement proche, le successeur et remplaçant du Uno.

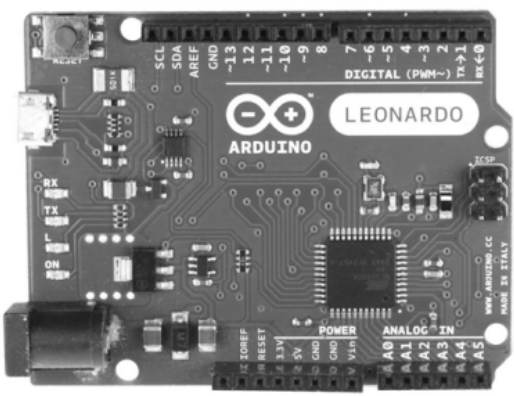

Figure 7.13 – L'Arduino Leonardo (doc. Arduino).

Même s'il lui est quasiment compatible à 100 %, si ce n'est quelques modifications d'ordre logicielles que nous verrons dans un instant, l'Arduino Leonardo utilise tout d'abord un microcontrôleur différent de l'ATmega328 du Uno, puisque c'est un ATmega32U4 plus récent. Le choix de ce processeur s'explique notamment par le fait qu'il intègre une interface USB en mode natif ce qui rend désormais inutile l'ATmega16U2 auquel était dévolu ce rôle sur les cartes Arduino Uno.

Hormis ces différences liées au processeur, l'Arduino Leonardo intègre les modifications faites sur l'Arduino Uno Révision 3 c'est-à-dire l'allongement des connecteurs avec la présence de IOREF, de SDA et de SCL. Son brochage est donc rigoureusement identique à celui du Uno Révision 3 vu au chapitre 1.

Trois autres différences cosmétiques entre le Uno Révision 3 et le Leonardo sont toutefois à mentionner :

– Le connecteur USB du Uno devient un connecteur micro USB moins encombrant.
– Les différentes LED du Uno (Tx, Rx, On et celle de la patte D13) sont désormais regroupées et bien alignées entre le connecteur micro USB et le jack d'alimentation, ce qui permet de bien les voir même si un shield est enfiché.
– Le processeur ATmega32U4 est en boîtier CMS et est donc soudé sur la carte, alors que l'ATmega328 de certaines versions de Uno était en boîtier DIL classique, enfiché dans son support. Il pouvait donc être retiré si nécessaire, ce que l'on peut être amené à faire lorsqu'on utilise l'Arduino comme programmateur de microcontrôleur Atmel comme nous le décrivons dans notre ouvrage *Arduino, Applications avancées*.

Ceci étant précisé, les principales différences entre l'Arduino Uno et l'Arduino Leonardo se situent au plan logiciel, que ce soit en phase de programmation ou d'exploitation, et sont essentiellement causées par l'absence de processeur spécifique pour gérer l'interface USB – série entre le PC et l'Arduino. Voici sommairement quelles en sont les conséquences.

Le Leonardo ne disposant pas d'un processeur spécifique pour gérer sa connexion *via* le port USB avec le PC, cela signifie que le port série qu'il utilise est virtuel, tant côté PC comme c'était déjà le cas avec le Uno, que côté Arduino, ce qui est nouveau. Cela implique que chaque fois que la carte Arduino subit un reset, ce port virtuel disparaît ce qui coupe nécessairement la connexion avec le PC, ce qui n'était pas le cas avec les autres Arduino.

Contrairement aux autres Arduino également, aucun reset de la carte n'est réalisé lorsque le port série est ouvert côté PC pour communiquer avec le Leonardo. Cela implique qu'il est possible de ne pas voir un certain nombre de données envoyées *via* le port série du Leonardo suite, par exemple, à l'ouverture du moniteur série de l'environnement de développement, en particulier les éventuelles données envoyées pendant la partie `setup()` d'un programme.

Pour y remédier, il suffit d'ajouter, au début de ce `setup()` la ligne suivante :

```
while (!serial) ;
```

© Dunod – La photocopie non autorisée est un délit.

On est assuré comme cela qu'aucune communication *via* le port série ne peut avoir lieu tant que ce dernier n'est pas ouvert.
Du fait de la dissociation du port série et du port USB permise par l'intégration d'une interface série dans le Leonardo, la classe `Serial` correspond désormais au port série virtuel créé par l'interface USB et non plus au port série matériel accessible *via* les lignes Tx (D1) et Rx (D0) de l'Arduino. Pour accéder au port série matériel du Leonardo il faut donc désormais utiliser `Serial1`.
En raison de l'absence d'un processeur intermédiaire pour assurer l'interface entre l'Arduino et le port série virtuel du PC, la communication *via* ce dernier est beaucoup plus rapide avec le Leonardo qu'avec un Arduino classique. De ce fait, il est possible, dans certaines configurations, de saturer le tampon du port série côté PC. Si tel était le cas lors de la répétition trop rapide d'une suite de `Serial.println()` par exemple, il suffirait d'ajouter une très légère temporisation à ce niveau de la façon suivante :

```
…
Serial.println(xxx) ;
Delay(1) ;
…
```

Ceci étant précisé, sachez que, comme tous les exemples de programmes et toutes les applications présentés dans cet ouvrage ont été développés et testés sur un Arduino Uno Révision 2 ou Révision 3, elles devraient aussi fonctionner avec un Arduino Leonardo sous réserve d'ajouter la ligne :

```
while (!serial) ;
```

pour celles d'entre elles qui utilisent le moniteur de port série.

7.4.2 Les Arduino Due, Yun et Galileo

Ces trois versions d'Arduino, même si elles restent compatibles de l'Arduino Uno de base pour ce qui est de certains shields et de l'environnement de développement, sont résolument tournées vers la puissance de programmation.
Le bon vieil ATmega328 de base y est ainsi remplacé, selon le cas, par un Atmel SAM3X8E ARM Cortex M3 pour l'Arduino Due ou bien encore par un Intel Quark SoC X1000 pour ce qui est du tout récent Galielo, en phase de pré-commercialisation au moment où ces lignes sont écrites.
Même si ce que vous avez appris dans cet ouvrage reste parfaitement valable et utilisable avec ces nouveaux Arduino, particulièrement en ce qui concerne l'interfaçage avec le monde extérieur, de nombreuses possibilités de programmation nouvelles voient le jour tel que, par exemple, le support de l'environnement Linux avec l'Arduino Yun.
Les découvrir toutes nécessite donc de très nombreux développements qui feront certainement l'objet d'un nouvel ouvrage…

WEBOGRAPHIE

Il existe aujourd'hui sur Internet d'innombrables sites consacrés à l'Arduino. Beaucoup ne sont que des sites commerciaux qui proposent les cartes Arduino de base et/ou des shields associés. D'autres sont très souvent des sites d'amateurs qui décrivent les réalisations qu'ils ont pu faire autour de l'Arduino. D'autres enfin mettent à disposition des bibliothèques spécifiques qui peuvent couvrir les domaines les plus divers.

Tenir à jour un répertoire de ces sites, dans le cadre d'un ouvrage imprimé de surcroît, est tout bonnement impossible tant ils apparaissent, mais aussi hélas disparaissent, rapidement. Nous ne vous proposons donc ci-dessous que les adresses des sites utilisés, pour une raison ou pour une autre, dans le cadre de cet ouvrage. Comme vous pourrez le constater si vous les consultez, cela couvre déjà la majorité des besoins.

– **Adafruit** – Fabricant et distributeur de nombreux shields et du Boarduino présenté au chapitre 7.

  http://www.adafruit.com

– **Arduiniana** – Nombreuses bibliothèques parfois très originales.

  http://arduiniana.org

– **Arduino** – Site officiel consacré à l'Arduino et, à ce titre, 1er site à consulter.

  http://www.arduino.cc

– **Electronic Kits Zone** – Distributeur de shields dont le LCD-Keypad Shield présenté au chapitre 3.

  http://www.ekitszone.com

– **Emartee** – Distributeur de shields dont l'Arduino Sensor Shield et ses « Bricks » présentés au chapitre 3.

  http://www.emartee.com

– **Freeduino** – Répertoire de « toutes » les ressources Internet consacrées à l'Arduino et à ses dérivés. Fichiers des différentes versions de Freeduino.

  http://www.freeduino.org

– **Freetronics** – Fabricant et distributeur du TwentyOne présenté au chapitre 7.

  http://www.freetronics.com

© Dunod – La photocopie non autorisée est un délit.

- **Lextronic** – Distributeur officiel en France des modules Arduino officiels, des shields Sparkfun et de divers autres shields.

  http://www.lextronic.fr

- **Miles Burton** – Bibliothèque destinée aux capteurs de température 18x20 de Dallas.

  http://milesburton.com

- **PJRC** – Bibliothèque de Jim Stud destinée aux circuits à bus un fil.

  http://www.pjrc.com/teensy/td_libs_OneWire.html

- **Seeed Studio** – Distributeur de shields dont l'Arduino Sensor Shield et ses « Bricks » présentés au chapitre 3.

  http://www.seeedstudio.com

- **Solarbotics** – Distributeur (entre autres) du Freeduino présenté au chapitre 7.

  http://www.solarbotics.com

- **Sparkfun** – Fabricant et distributeur de très nombreux shields utilisés dans l'ouvrage.

  http://www.sparkfun.com

- **Site de Ch. Tavernier** – Site de l'auteur de cet ouvrage.

  http://www.tavernier-c.com

INDEX

Symboles

^ 45
- - 40
; 26
! 44
{ 27
*/ 26
/ 26
/* 26
// 26
& 45
&& 44
% 40
++ 40
<< 45
= 39
= = 39
>> 46
| 45
|| 44
~ 45
74HC595 77, 155
/SS 171

A

abs 41
accolades 27
AD5171 165
Adafruit Industries 203, 207
addition 39
afficheur
  à LED 104, 156, 173
  alphanumérique 81, 109
  LCD 82, 109
  multiplexé 106
  sept segments 104, 156, 173
alimentation 2
ampoule au mercure 86
analogRead() 41, 42, 97, 124
analogReference() 57, 118
analogWrite() 59, 129, 142
appel général 162
Arduiniana 207
Arduino 1
  alimentation 6
  dimensions 5
  Ethernet Shield 187
  Fio 195
  horloge 6
  langage évolué 18
  LilyPad 199
  Mega 1280 11
  Mega 2560 11
  mémoire 7
  Mini 192
  Nano 194
  Pro 196
  Pro Mini 198
  reset 6
  Sensor Shield 80
  servo 133
Arduino and Breadboard Holder 76
Arduino Duemilanove
  dimensions 5
Arduino Uno 5
  dimensions 5
  schéma 10
Ardumoto Motor Shield 144
array 32
array of strings 34
ASCII 68, 69, 112
assembleur 15
astérisque 34
ATmega
  168 5
  328 5, 7
  2560 11
  ATmega8U2 10
Atmel 4
attach() 133
attachInterrupt() 66
autoscroll() 114
available() 164
AVR 4, 18

B

begin() 113, 163, 171, 172
beginTransmission() 163

© Dunod – La photocopie non autorisée est un délit.

bibliothèque 28
keypad 94
LiquidCrystal 113
OneWire 179
Servo 133
SPI 172
Stepper 140
Wire 162
bitClear() 65
bitRead() 64
bitSet() 65
bitWrite() 64
Boarduino 203
boolean 31, 37
bootloader 7
boucle 49, 50
break 49, 51
bricks 80
brochage des connecteurs 10, 13
bus « un fil » 176, 177
buzzer , 77
byte 30

C

C 18, 25
C++ 18, 25
capteur
résistif 119
tout ou rien 86
carte SD 189
case 48, 49
CCITT V24 151
CdS 118
chaînes de caractères 32
char 33
circuits logiques TTL 2, 8, 38
classe 33
clavier 90
analogique 94
en matrice 90, 93
téléphonique 92
clear() 114
code ASCII 68, 69, 112
COM 67
commutateurs 86
compilateur 16
condition
d'arrêt 161
de départ 161
const 28
constrain 41
continue 52
convertisseur analogique/digital 9, 57, 117
cos 42
CPHA 172
CPOL 171
CRC 181
crc8() 181
CTN 118

D

Dallas 177
Danger Shield 77
Darlington 102
décalage
à droite 46
à gauche 45
décrémenter 40
define 27
delay() 54
delayMicroseconds() 54
detach() 134
detachInterrupt() 66
digitalRead() 56
digitalWrite() 55
division 39
entière 40
donnée
array 32
boolean 31
byte 30
char 33
double 31
float 31
indexée 32
int 30
long 30
unsigned char 33
unsigned int 30
unsigned long 31
word 30
données logiques 31
do/while 51
DS18B20 182
DS18x20 182
DS1307 167

E

éditeur de texte 17
EEPROM 11, 12
Electronic Kits Zone 207
Eleven 201
else 48
Emartee 207
end() 173
endTransmission() 164
entrée parallèle 86
entrées analogiques 12, 13, 55, 117
entrées/sorties 8, 85
numériques 8, 12, 55
environnement de développement 19
Ethernet 189
Ethernet Shield 187
ET logique 44

F

false 37
Fio 195
float 31
fonction 27
for 50, 52
Freeduino 202, 207
Freetronics 201, 207

FT232 10
fusible réarmable 6

G

goto 53
gradateur
  de puissance 126
  de tension 126

H

haute impédance 38
haute tension 97
HIGH 55, 56
highByte() 64
home() 114
horloge 2
  temps réel 167
hystérésis 120

I

I2C 9, 12, 157, 158
ICSP 4, 73
if 43, 47
if/else 48
ILS 86
include 28
incrémenter 40
index 32
INPUT 38, 55
int 30
Internet 186
interruptions 9, 12, 65
interrupts() 66
ISP 4

L

L298 143
langage
  évolué 15
  machine 14
LCD - Keypad Module 82
LDR (Light Dependent Resistor) , 77
LED 99
leftToRight() 114
liaison série
  asynchrone 147
  synchrone 154
ligne
  de code 26
  de commentaire 26
LilyPad 199
LiquidCrystal() 113
listing source 16
long 30
loop 29
LOW 55, 56
lowByte() 64

M

map() 41, 124
masse 39
max 41
MAX
  220 153
  232 151
  233 151
  249 153
  7219 175
Maxim 177
Mega 1280 11
Mega 2560 11
mémoire
  Flash 7, 11, 12
  morte 1, 3
  vive 1, 7
microcontrôleur 1
microprocesseur 1
micros() 54
Miles Burton 208
millis() 54
min 41
Mini 192
Mini USB Adapter 193, 198
MISO 171
MJ 3001 102
MLI 58, 126
mnémoniques 14
MOC 3040 103
MOC 3041 103
mode
  4 bits 110, 111
  8 bits 111
mode() 173
modulation de largeur d'impulsions 9, 12, 58, 126
modulo 40
moniteur de port série 23, 67
MOSI 171
moteurs 102, 130
  à courant continu 142
  bipolaires 137
  pas à pas 130, 137
  unipolaires 137
multiplexage 106
multiplication 39

N

NACK 165
Nano 194
next 50
noAutoscroll() 114
noInterrupts() 66
nombres
  aléatoires 62
  négatifs 46
NON logique 44
noTone() 60

O

OneWire() 180

© Dunod – La photocopie non autorisée est un délit.

opérateur
  arithmétique 39
  d'assignation 39
  de comparaison 39
ou exclusif logique 44
OU logique 44
outil de développement 14
OUTPUT 38, 39

P

photocoupleurs 100
photorésistance 118
photo triac 103
pinMode() 54, 55
pointeurs 34
point-virgule 26
Polyfuse 6
port série 67
port série virtuel 20
potentiomètre numérique 165
poussoir de reset 72, 77
poussoirs 86
pow 42
print() 115, 150
println() 150
Pro 196
programmateur 17
programmation en circuit 4
Pro Mini 198
ProtoShield 73, 74
puissance 42
pulseIn() 61
Pulse Width Modulation 126
PWM 12, 58, 126

R

racine carrée 42
RAM 11, 12
random() 63
randomSeed() 62
rapport cyclique 59, 127
read() 134, 149, 150, 180, 181
rebondissements 88
receive() 164
registre à décalage 154
relais 99
relais statique 103
requestFrom() 164
reset 3
  automatique 6
reset() 180
reset_search() 180
résistance à coefficient de température négatif 118
reste 40
return 52
rightToLeft() 114
ROM 3
roues codeuses 86
RS 232 147, 150
RxD 153

S

SCK 171
SCL 159, 163
scope 35
scrollDisplayLeft() 114
scrollDisplayRight() 114
SDA 159, 163
SDI 171
SDO 171
search() 180
secteur EDF 97, 103
seed 62
Seeed Studio 208
select() 181
send() 163
Sensor Shield 80
Serial () 147
Serial.available() 68
Serial.begin() 67
Serial.end() 68
Serial.flush() 68
Serial.print() 69
Serial.println() 70
Serial.read () 68
Serial.write() 70
serveur Internet 188
servo de radiocommande 130, 135
servomoteurs 130
setCursor() 114
setDataMode() 173
setSpeed() 140
setup 28
shield prototype 72
shields 6, 8, 71
  évolués 185
shiftOut() 61, 156
sin 42
sizeof 37
sketches 18
skip() 181
SoftwareSerial() 149
Solarbotics 208
soustraction 39
Sparkfun 74, 196
SPI 9, 12, 61, 168, 170
sqrt 42
static 36
Stepper() 140
steps() 140
string 33
String 33, 35
switch 48, 49

T

tableau 32
  de chaînes de caractères 34
tan 42
temps de conversion 58
tension de référence 117

TMP36 , 122, 77
tone() 60
transfer() 173
true 37
TwentyTen 201
TxD 153

U

UART 147
ULN 2803 , 101
unité centrale 1
unsigned char 33
unsigned int 30
unsigned long 31
USART 147

V

V24 151
virgule flottante 31
void 31
volatile 36

W

while 51
word 30
write() 134, 180, 181
writeMicroseconds() 134

X

Xbee 195

© Dunod – La photocopie non autorisée est un délit.

Pour l'éditeur, le principe est d'utiliser des papiers composés de fibres naturelles, renouvelables, recyclables et fabriquées à partir de bois issus de forêts qui adoptent un système d'aménagement durable.

En outre, l'éditeur attend de ses fournisseurs de papier qu'ils s'inscrivent dans une démarche de certification environnementale reconnue.