Partie 1

Introduction à la conception par l'exemple

Introduction

Dans cette partie, nous allons considérer quatre modèles de programmation de systèmes embarqués. Les trois premiers sont simples, mais limiteront notre potentiel à effectuer des traitements rapidement dès lors qu'une partie des traitements peut durer longtemps. En effet, les trois premiers modèles de programmation ne permettent que sous certaines conditions d'interrompre un traitement en cours. Le dernier modèle de programmation utilise le multitâche, qui se base sur la préemption, permettant d'interrompre un traitement pendant une durée, même importante, afin de consacrer les ressources de calcul à un autre traitement.

Au passage, nous introduirons des formalismes de description d'un système à différentes phases de la définition de celui-ci. En effet, autant lorsqu'on est en équipe réduite de développement, sur un projet court, il est possible de mener un projet à bien quasiment directement en le programmant, autant lorsque le projet est plus long, et/ou lorsque l'équipe le développant est de taille relativement importante, il est primordial de se mettre d'accord dès le début sur le système à réaliser.

Chapitre 1 Présentation du système exemple

Introduction

Afin d'introduire le contexte logiciel et matériel d'un système embarqué ou cyber-physique, nous prenons un petit exemple basé sur une carte Arduino Uno®. Le but n'est pas de présenter la programmation sur microcontrôleur, mais d'en utiliser un exemple concret afin de remonter par la suite aux méthodes de spécification et conception qu'il est conseillé d'utiliser pour arriver à du logiciel embarqué sûr.

Qui dit embarqué dit utilisation de capteurs et actionneurs, les grandes familles de capteurs et actionneurs sont introduites, en particulier les moyens de communication entre le calculateur et ceux-ci. Ces moyens de communication donnent de l'importance à la façon dont tout calculateur représente l'information, en binaire. Nous voyons donc la représentation des données, et les principaux pièges tendus à l'ingénieur par le fait qu'elle soit de taille finie.

Objectifs

- **Avoir** des bases d'achitecture matérielle
- **Connaître** la représentation de l'information
- **Apprendre** de bonnes pratiques de programmation
- **Savoir** ce qu'est une fonction

Plan

1. Montage expérimental
2. Programme de contrôle

Le système que nous voulons réaliser est un système de ventilation : nous mesurons la température, et lorsque celle-ci dépasse une température de consigne, un ventilateur est commandé proportionnellement à la différence de température entre la mesure et la consigne. Nous souhaitons afficher sur une console différentes informations mesurées, comme la température, la vitesse du ventilateur, et la valeur de commande envoyée à celui-ci.

La température de fonctionnement devrait se situer entre 0 et 100 °C.

1 Montage expérimental

Pour la mesure de température, nous choisissons une sonde analogique de type TMP36, dont la plage de fonctionnement est compatible avec notre température de fonctionnement.

TMP36 : de la marque Analog Devices®, les capteurs de température basse tension TMP36 sont des capteurs analogiques. Ceux-ci sont alimentés en voltage faible (2,7 V à 5,5 V) et consomment peu de courant (~50 µA). Ils fournissent en sortie une tension linéaire, proportionnelle à la température mesurée. À la température de –50 °C, le capteur fournit 0 V en sortie, puis la tension augmente de 10 mV par °C. Il nous faut lire une tension d'entrée variant entre 0 V et 1,75 V, correspondant linéairement à la température lue de –50 °C à +125 °C. Il offre une précision de +/– 1 à 2 °C, suffisante pour notre application, et pour un coût raisonnable (1,50 € si acheté à l'unité, et moins de 50 centimes en grandes quantités).

Tension analogique *versus* signal numérique

Les signaux habituellement utilisés en informatique peuvent être de deux natures : **analogiques** ou **numériques**. Un signal analogique se caractérise par un état instantané évoluant dans le domaine continu, entre une valeur minimale et une valeur maximale. Typiquement entre 0 V et 5 V ou entre 0 V et 10 V, ou encore entre deux seuils spécifiques, comme pour la sonde TMP36 le signal analogique de sortie varie entre 0 V et 1,75 V. Un signal numérique ne peut prendre qu'un ensemble limité de valeurs : on parle aussi parfois de signal **discret**. Dans la plupart des cas, le signal numérique est bivalent : il ne peut prendre que deux états, un état haut et un état bas (par exemple 0 V et 5 V ou bien 0 V et 12 V). Dans ce cas, l'information est codée en utilisant le temps : par exemple sur un PWM, le rapport entre durée du front haut et durée du front bas encode une information, ou bien sur la liaison série RS-232C l'état pendant un temps bit (dont la durée est égale à l'inverse du débit) permet de représenter une information binaire (pour un signal bivalent transmis à 9 600 bauds, chaque bit est encodé par l'envoi d'un signal haut ou bas durant un 9 600e de seconde).

Le ventilateur sera un ventilateur de processeur de type quatre fils, récupéré sur un ordinateur. Ce ventilateur est conforme à la spécification Intel (septembre 2005, révision 1.3).

Ventilateur de processeur quatre fils : possède deux connecteurs pour l'alimentation (12 V et masse), un connecteur pour la consigne de vitesse, acceptant un signal PWM à 25 kHz, et un connecteur en drain ouvert envoyant une impulsion PWM par demi-tour du ventilateur.

Signal PWM (Pulse Width Modulation)

La modulation par largeur d'impulsion consiste à faire varier un signal discret entre les états hauts et bas pendant des durées définies pour transmettre des données ou de la puissance. Un signal PWM se caractérise par son rapport cyclique (durée de l'état haut par rapport à la durée de l'état bas) et sa fréquence, c'est-à-dire l'inverse de sa période, constituée de la somme de la durée de l'état haut et de l'état bas (voir figure 1.1). On retrouve ce type de signaux en modélisme pour commander des servomoteurs ou pour les transmissions de la radio commande. Le PWM peut aussi être utilisé pour « simuler » un signal analogique entre la valeur basse et la valeur haute, car après filtrage (passe-bas), le rapport cyclique donne un signal qui peut évoluer de façon continue entre la valeur haute (rapport cyclique

de 1) et la valeur basse (rapport cyclique de 0 ou proche de zéro). C'est une façon de générer, à bas coût, une tension analogique à partir d'un générateur de signaux numériques.

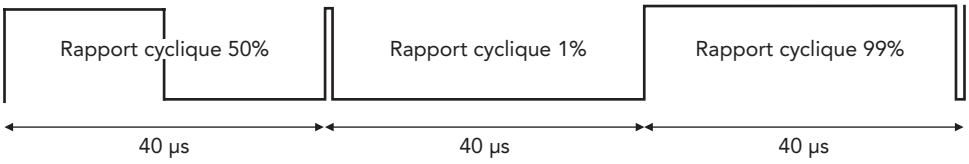

Figure 1.1 – Signal PWM de fréquence 25 kHz.

La figure 1.2 illustre la manière dont une sortie PWM permet de reproduire un signal d'allure sinusoïdale à partir d'une modulation de largeur d'impulsion dont le rapport cyclique suit la valeur d'une sinusoïde positive de fréquence 1 kHz. On observe que le signal résultant comporte un certain nombre de paliers qui dépendent du suréchantillonnage de la sortie PWM (nombre de cycles PWM par période d'une milliseconde, ici fixé à huit). On observe également un déphasage entre le sinus de sortie et le signal souhaité, lié à la présence d'un filtre passe-bas (d'ordre 2 dans l'exemple). Le filtre passe-bas supprime les transitoires du signal PWM (hautes fréquences) pour ne laisser passer qu'une moyenne du signal entrant (basses fréquences). Cette moyenne est faite sur une durée caractéristique du filtre, liée à sa fréquence de coupure et à son facteur de qualité. Il faut donc trouver un compromis entre un bon filtrage des hautes fréquences (bruit), le retard de phase induit et la complexité du filtre (ordre).

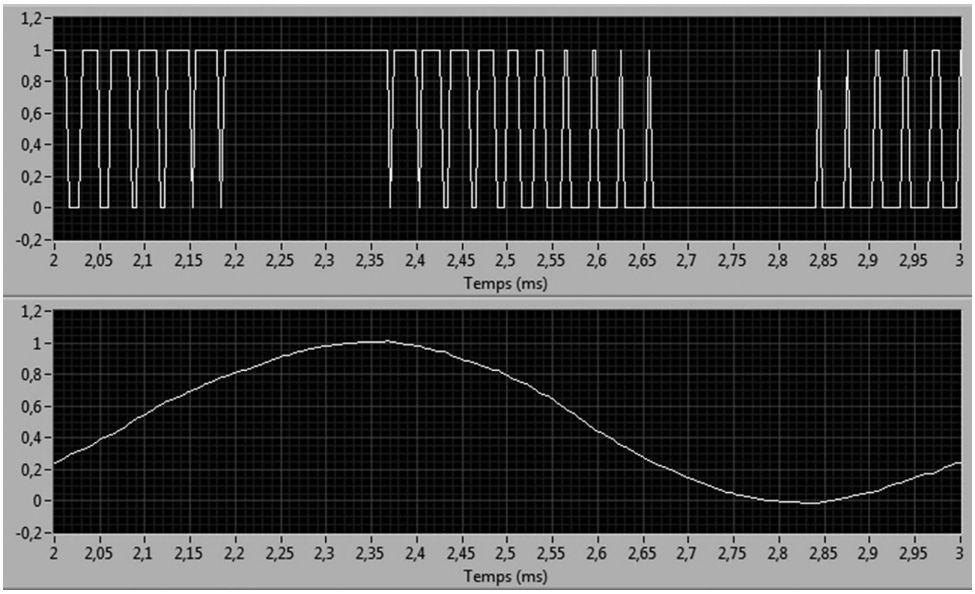

Figure 1.2 – Génération d'un signal sinusoïdal à 1 kHz à l'aide d'un PWM à 8 kHz et d'un filtre passe-bas d'ordre 2.

Il nous faut donc un système de contrôle permettant de lire une entrée analogique (capteur de température), une entrée PWM (rotation du ventilateur), et d'écrire du PWM en sortie à une fréquence de 25 kHz (commande du ventilateur). Une connexion de type série ou USB permettrait de réaliser l'affichage initialement sur un PC.

Nous choisissons une carte microcontrôleur Arduino Uno®, munie d'entrées-sorties, parmi lesquelles nous pouvons trouver notamment six entrées analogiques, et une douzaine d'entrées-sorties numériques. Ces dernières nous permettront de lire et de générer des signaux PWM.

Arduino Uno : carte matérielle embarquant un microcontrôleur huit bits ATmega328P à 16 MHz équipé de 32 ko de mémoire flash. La carte intègre notamment 14 entrées/sorties numériques (dont six sont compatibles PWM), six entrées analogiques multiplexées, et une liaison série.

Le montage expérimental[1] est donné sur la figure 1.3. La sortie analogique (fil orange) de la sonde TMP36 est reliée à l'entrée analogique `A0` du microcontrôleur. Pour son alimentation, elle doit également être connectée à la masse et à la sortie 5 V du microcontrôleur. La commande PWM (fil bleu) du ventilateur est liée à la broche 3 `INT1` du microcontrôleur, alors que la sortie tachymètre (fil vert), lié à la broche 2 `INT0`, est montée avec une résistance de tirage de 10 kΩ, de façon classique pour un circuit en drain ouvert (voir encadré). Contrairement à la sonde, l'alimentation du ventilateur se fait en 12 V (fils jaune et noir).

Le microcontrôleur est quant à lui directement alimenté par le cordon USB relié au PC sur lequel nous effectuerons le développement, et sur lequel l'affichage se réalisera.

Un peu d'électronique pour les nuls

Montage à drain ouvert : le signal PWM tachymètre du ventilateur n'est pas généré directement par le ventilateur, en réalité, la sortie tachymètre du ventilateur (fil vert) est branchée en sortie d'un transistor. Selon que le transistor est ouvert ou fermé, la sortie est donc soit reliée à la masse, soit à l'alimentation. On utilise donc une résistance de tirage, typiquement assez élevée, de façon à limiter le courant et ainsi protéger l'entrée de l'Arduino, puisque selon la loi d'Ohm, ce courant est proportionnel au rapport de la tension d'alimentation (ici 5 V) sur la résistance (ici 10 kΩ).

Plaque d'essai : une plaque d'essai permet de brancher facilement des composants électroniques. Sur la figure 1.3, les connecteurs des deux lignes supérieures et des deux lignes inférieures sont reliés par ligne. Les lignes A, B, C, D, E (respectivement F, G, H, I, J) de connecteurs sont reliées par colonne. Ainsi, par exemple, A1, B1, C1, D1, E1 sont reliés.

1. Les figures de montages électroniques sont réalisées à l'aide du logiciel Fritzing (*http://fritzing.org*).

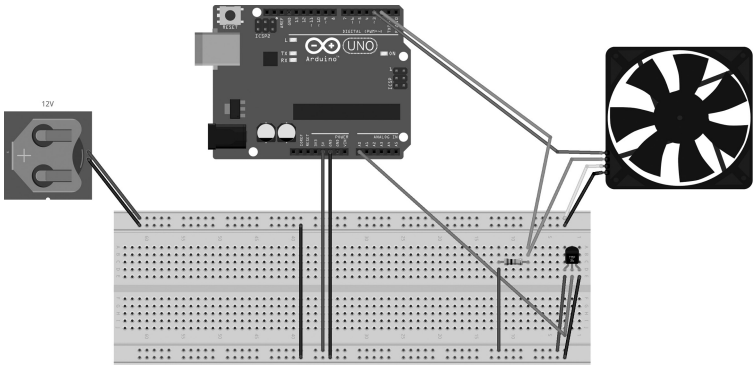

Figure 1.3 – Montage expérimental du système de ventilation.

2 Programme de contrôle

Arduino propose un environnement de développement (**IDE** – *Integrated Development Environment*) relativement simple d'utilisation, mais qui n'offre que des fonctionnalités basiques de développement.

Nous utiliserons le langage C, avec l'interface de programmation (**API** – *Application Programming Interface*) native d'Arduino. Le principe d'un programme C développé pour Arduino est le suivant : à la place d'un sous-programme principal nommé `main()`, un programme Arduino possède un point d'entrée sous forme d'un sous-programme nommé `setup()`. Ce sous-programme est appelé juste après que la plateforme soit initialisée, et doit initialiser tout ce qui doit l'être pour l'application utilisateur. Ensuite, le sous-programme `loop()` est appelé en boucle. Il faut noter que ce système de programmation (`setup()` puis `loop()` appelé en boucle) est spécifique à l'API Arduino, comme nous le verrons en utilisant d'autres API dans la suite de l'ouvrage.

Exceptionnellement, nous travaillerons directement sur le programme C, sans d'abord le spécifier fonctionnellement ou exprimer l'architecture logicielle et matérielle du système. Cela nous permettra dans la suite immédiate d'introduire simplement des éléments d'un langage dédié (que l'on appelle un **DSL** pour *Domain Specific Language*) à la spécification et la conception de systèmes. Ce langage est outillé par le logiciel **Capella** (*https://www.polarsys.org*). Capella a été initialement développé par et pour la société Thales, qui l'a ensuite ouvert au public *via* la fondation Polarsys. Il s'agit d'un logiciel libre, publié sous licence EPL (*Eclipse Public Licence*). Capella met en œuvre la méthode de spécification et de conception basée sur les modèles **Arcadia**, dont Thales et le consortium Clarity font la promotion. Cet ouvrage ne couvre pas l'ensemble de la méthode Arcadia, qui vise à intégrer tous les acteurs impliqués dans la conception de systèmes industriels complexes (ingénieurs commerciaux, analystes, programmeurs, architectes systèmes, instrumentiers, etc.). Nous utiliserons uniquement

un sous-ensemble des diagrammes proposés par Capella, car l'outil est à ce jour, la solution ouverte et libre basée modèle la plus aboutie. Cependant, Capella ne fournit que peu d'éléments de modélisation pour la projection des fonctions sur le système embarqué. Nous utiliserons donc un autre DSL, en l'occurrence AADL, outillé par le logiciel libre **Osate** (*http://osate.org/*), disponible sous licence CPL (*Common Public License*), pour cette partie. Enfin, même si Capella propose des machines à états à la UML, leur implémentation étant partielle à l'heure où ces lignes sont écrites, nous utiliserons un outil spécifique pour les statecharts : **Yakindu SCT** (*https://www.itemis.com/en/yakindu/state-machine/*), sous licence propriétaire, mais gratuit pour un usage non commercial. Les trois outils sont basés sur la plateforme libre Eclipse (*https://www.eclipse.org/*).

Les programmes et tout support numérique utiles sont disponibles sur le site internet *http://www.dunod.com*. Il est conseillé à ce niveau de télécharger le matériel numérique afin de pouvoir compiler, tester, et potentiellement modifier les programmes.
Il existe de nombreuses fonctions spécifiques à chaque API. Ainsi, par exemple, Arduino propose de nombreuses fonctions comme `Map`, qui permet de faire une projection, ou `Constrain`, qui permet de contraindre une valeur dans un intervalle. Ce livre n'a cependant pas été écrit pour une API ou un microcontrôleur en particulier. Par conséquent, nous limiterons en général l'utilisation d'une API au minimum, et reprogrammerons les fonctions utilitaires.
Les auteurs de cet ouvrage ne sont pas liés au développement ou à la promotion de Capella, au moment d'écrire ces lignes et de choisir une méthode support à l'ouvrage. Notre choix s'est porté sur ce langage et outil après de multiples tentatives d'exprimer ce dont nous avions besoin à l'aide de plusieurs langages et outils. La version précédente de l'ouvrage utilisait SysML, cependant les *FlowPort* ayant été retirés des dernières versions de la norme SysML, notre principale motivation pour le choix de ce langage n'est plus. La première version de l'ouvrage, datant de 2005, utilisait SA-RT, qui n'est plus outillé, mais on retrouve dans Capella de nombreux concepts que l'on trouvait notamment dans SA, et aussi dans SA-RT.

2.1 Définition des constantes importantes et des variables globales du programme

Il est d'usage de définir sous forme de constantes les éléments d'entrées-sorties que nous utiliserons afin de n'y faire référence qu'une seule fois (à la définition des constantes) dans le programme. En effet, cela nous rend moins dépendants du matériel, et des connexions effectuées. Nous avons ici trois connecteurs d'entrée-sortie utilisés. Nous les définissons comme des constantes entières. Ainsi, `VentilateurPin` est une constante entière (mot-clé `const`), à laquelle on affecte le **littéral** `3`. L'entrée analogique 0 bénéficie d'une constante `A0` prédéfinie dans l'API Arduino.

```
const int TemperaturePin = A0;
const int VentilateurPin = 3;
const int tachymetrePin = 2;
```

Représentation de l'information

Toute information numérique est représentée en binaire, c'est-à-dire en base 2. En base décimale, que nous utilisons tous les jours, la base est 10, ce qui nous laisse les dix chiffres 0 à 9 pour représenter chaque multiplicateur de puissance de 10. Ainsi, le nombre 123 se lit $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$, et est donc représenté par 123 en base décimale. En binaire, il n'y a que deux chiffres, 0 et 1. Un chiffre binaire s'appelle un bit, pour *BInary digiT*. Les bits permettent de représenter le nombre 123 comme $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$, ce qui se lit 1111011 en base binaire. Afin de distinguer les bases, nous utiliserons en préfixe la lettre « b » pour les nombres donnés en binaire. Le nombre 123 se représente donc b1111011 en binaire.

Représentation finie de l'information

La mémoire d'un calculateur est bornée. Les calculs sont effectués par l'unité arithmétique et logique du processeur. Les opérandes et les résultats sont stockés dans des « cases mémoire » appelées registres qui sont de taille finie. Leur taille définie celle d'un **mot mémoire**, généralement multiple de huit bits. Un groupe de huit bits s'appelle un **octet**. Les architectures des ordinateurs utilisent des mots mémoire d'un à huit octets (64 bits), par conséquent, il en va de même pour les langages de programmation qui proposent d'utiliser des variables numériques dont la taille peut varier entre un et huit octets. Sur un octet, on peut représenter des valeurs allant de b00000000 à b11111111, ce qui correspond à des valeurs allant de 0 à 255 (soit $2^8 = 256$ valeurs différentes). Nous voyons donc différents problèmes qui ont dû être réglés : la représentation des nombres signés (négatifs ou positifs), la représentation des nombres fractionnaires, des nombres réels, et les problèmes de débordement. Nous aborderons ces points plus tard dans l'ouvrage lorsque nous nous trouverons face à ces problèmes.

Le fait qu'Arduino lance un sous-programme `loop()` en boucle nous oblige, lorsqu'une variable doit être mise à jour en fonction de son ancienne valeur dans cette boucle, soit à la déclarer en variable locale statique, soit à la déclarer comme une variable globale : dans les deux cas, cette variable est mise dans le tas et pas dans la pile du sous-programme, ce qui permet de la conserver même après la terminaison du sous-programme pour un usage ultérieur.

Rémanence des variables, pile et tas

Les variables déclarées hors de tout sous-programme sont en général globales, c'est-à-dire visibles de tout sous-programme. Elles sont, de plus, conservées au même endroit de la mémoire pendant toute la durée du programme. Au contraire, les variables déclarées dans un sous-programme sont temporaires, elles n'existent que pendant la durée du sous-programme.

Ainsi, dans notre sous-programme `loop()`, si nous déclarons localement (c'est-à-dire à l'intérieur du sous-programme) une variable température, celle-ci est perdue à la fin du sous-programme. Lors de la prochaine exécution du sous-programme `loop()`, c'est une nouvelle variable qui sera fraîchement créée, et elle ne contiendra pas la valeur de la variable de l'exécution précédente du sous-programme. Une variable locale peut être rendue persistante, en utilisant le mot-clé `static` lors de sa déclaration.

Ce comportement se comprend d'autant mieux lorsque l'on connaît le modèle mémoire des programmes. La mémoire d'un programme mono-tâche est (de façon réelle ou virtuelle) découpée en trois segments (voir figure 1.4). Le code (c'est-à-dire les instructions élémentaires du programme, représentées en binaire), supposé être statique, est mis dans le **segment de code** (certains systèmes d'exploitation le contraignent à être en lecture seule). Le **tas** contiendra les variables dont la durée de vie est égale à celle du programme, c'est-à-dire les variables globales, et les variables statiques de sous-programme. Le tas va aussi héberger la mémoire allouée dynamiquement (`new`, `malloc`, etc.). La **pile**, à l'instar d'une pile d'assiettes, est de nature « dernier arrivé, premier retiré » : sur une pile, on empile (on ajoute des assiettes sur une pile d'assiette, ou une donnée au sommet d'une pile) et on dépile (on retire l'assiette du sommet de la pile d'assiettes, ou une donnée du sommet d'une pile). Le fait d'empiler a pour effet d'augmenter la taille de la pile, et de faire monter son sommet (notons que comme la pile est représentée inversée sur la figure 1.4, le sommet descend, mais c'est un point de détail d'implémentation, cela ne change rien au comportement de la pile), alors que le fait de dépiler produit l'effet inverse. La pile est donc caractérisée par son contenu, mais seule l'adresse de son sommet est mémorisée. Lors de l'appel d'un sous-programme, on va commencer par empiler l'adresse de retour du sous-programme (c'est-à-dire l'adresse de l'instruction qui devra être exécutée après la terminaison du sous-programme) de façon à ce que le système sache ce qu'il devra faire à la fin du sous-programme. Ensuite, on empile tous les paramètres du sous-programme, ce qui permettra au sous-programme d'accéder à ses paramètres en

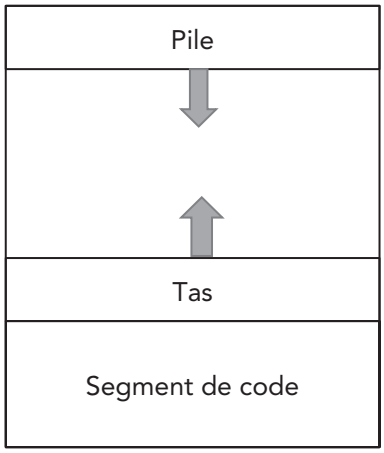

Figure 1.4 – Modèle mémoire classique d'un programme mono-tâche.

allant à un certain décalage du sommet de pile. Enfin, toutes les variables locales sont empilées. Ce qui est intéressant, c'est qu'à la fin du sous-programme, il suffit de dépiler (ce qui se traduit juste par le changement de l'adresse du sommet de pile) pour remettre la pile dans l'état dans lequel elle était avant appel du sous-programme, et reprendre l'exécution à l'adresse de retour. Ce mécanisme permet ainsi d'enchaîner les appels de sous-programmes très simplement, et même de faire facilement de la récursivité (sous-programme qui s'appelle lui-même, qui s'appelle lui-même, etc.).
En contrepartie de cette simplicité et flexibilité, toute variable locale est perdue (puisque dépilée) à la fin du sous-programme.

La sonde de température est assez sensible aux variations de tension, en effet, nous avons vu qu'une variation de 10 mV correspond à 1 °C. Une faible variation de tension liée à une erreur aura donc un impact important sur la température mesurée. Nous introduirons donc un filtre très simple de température : le filtre exponentiel de Poisson. Ainsi, si la première mesure de température est notée T_0, nous considérerons que la température considérée à la mesure courante est donnée par $T_i = (1 - \lambda) T_{i-1} + \lambda T$ où T_{i-1} est la température considérée à la mesure précédente, et T est la mesure effectuée. Le paramètre λ est un facteur d'oubli : lorsqu'il vaut 1, le filtre considère la mesure instantanée comme étant la température courante, alors que lorsqu'il a une valeur faible, les variations de température sont absorbées. Ce type de filtre lisse les mesures et réduit le bruit lié aux erreurs, mais en contrepartie diminue la réactivité du système.

Afin de programmer ce filtre très simple, nous avons besoin de conserver la dernière température lue, ce qui va être fait dans la variable `temperature`.

```
float temperature ;
```

Les flottants et la norme IEEE 754

Dans la plupart des systèmes, il est nécessaire de représenter des grandeurs fractionnaires. Bien entendu, la représentation étant de taille finie (sur un nombre fini d'octets), la représentation de nombres réels est impossible. On est donc amené à utiliser des représentations potentiellement arrondies des nombres fractionnaires. Là, il y a deux possibilités : lorsque l'on connaît parfaitement le domaine de variation du nombre, et la précision voulue, et on utilise des représentations en **virgule fixe**, à savoir des entiers avec unité non unitaire (typiquement fractionnaire). Nous verrons cette représentation lorsque nous parlerons des convertisseurs analogique-numérique. Cependant, si on dispose de peu d'informations concernant le domaine de variation du nombre, et/ou la précision voulue, on utilise les nombres à **virgule flottante**, plus simplement appelés **flottants**. Ces nombres, normalisés dans la norme IEEE 754, permettent de représenter des grandeurs avec une précision variable dépendant de la valeur du nombre. Si le nombre est petit, on souhaite en général une précision très importante, mais s'il est grand, on peut souvent se contenter d'une précision bien moindre.

Les flottants représentent donc les nombres en les décomposant à l'instar de la notation scientifique utilisée pour les nombres : par exemple un nombre pourrait s'écrire $4{,}3456 \times 10^{-7}$. Les flottants reprennent ce principe, mais en binaire. Ils existent en différentes tailles, mais les plus répandus sont les flottants **simple précision** sur 32 bits (`float`) et les flottants **double précision** sur 64 bits (`double`). Un flottant 32 bits est représenté tel que sur la figure 1.5 : le bit de poids fort est le bit de signe (1 pour moins, 0 pour plus), puis il y a huit bits représentant l'exposant, et 23 bits représentant la mantisse. À l'instar de la notation scientifique selon laquelle il y a un unique chiffre non nul avant la virgule, le flottant présente avant la virgule un chiffre non nul, qui est donc nécessairement 1. Puisqu'il est toujours égal à 1, ce bit est « caché » (c'est-à-dire qu'il n'est pas représenté) afin de gagner en domaine de représentation. Un nombre flottant se lit donc : $-1^{\text{signe}} \times 1{,}\text{mantisse} \times 2^{\text{exposant}}$.

Problème : on souhaite pouvoir représenter des grands et des petits nombres, on veut donc des exposants positifs et négatifs, mais il n'y a pas de signe à l'exposant. Dans la norme, l'exposant est dit « biaisé » : on l'interprète en lui soustrayant 127. Ainsi, un exposant à 127 se lit 0, et le plus grand exposant représentable est 254 – 127 = 127 alors que le plus petit représentable est 1 – 127 = –126. Nous verrons que les exposants représentés par 0 et 255 ont un sens particulier.

Problème : le 0 ne se représente pas, par conséquent, certaines valeurs binaires ont un sens particulier : ainsi, si la mantisse et l'exposant sont nuls, on considère que le nombre est nul (notons qu'il existe +0 et –0).

Problème : si la mantisse est non nulle, alors le plus petit nombre non nul en valeur absolue serait $1{,}000\ldots0001 \times 2^{-126}$. Afin de permettre une représentation fine autour de zéro, il existe donc une représentation dite dénormalisée (en opposition à la représentation dite normalisée présentée jusqu'ici). Si la valeur de l'exposant n'est constituée que de zéros, alors le bit caché est considéré comme valant 0 et l'exposant vaut –126 (comme lorsque l'exposant est représenté par 1). Ainsi le plus petit nombre non nul en valeur absolue est $0{,}000\ldots001 \times 2^{-126} = 2^{-149}$ ce qui vaut environ $1{,}4 \times 10^{-45}$.

Il existe aussi des représentations pour l'infini, et les erreurs (*Not a Number* ou NaN), ainsi qu'une arithmétique particulière prenant en compte les nombres, l'infini, et les erreurs.

La valeur maximale (255) de la représentation de l'exposant est réservée aux représentations de l'infini et des erreurs, par conséquent le plus grand nombre

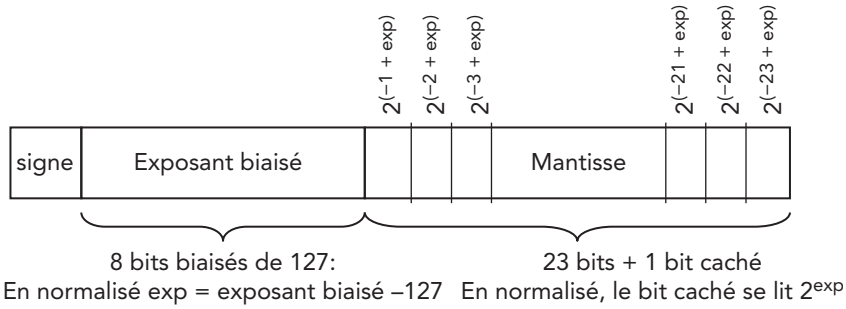

Figure 1.5 – Flottant 32 bits dans la norme IEEE 754.

normalisé a un exposant de 254 – 127 = 127. Le plus grand nombre représentable est donc 1,111…111 × 2^{127} ce qui vaut environ 3,4 × 10^{38}. Il faut cependant bien avoir en tête le fait qu'à cet exposant, deux nombres représentables successifs ont un écart de $2^{-23} \times 2^{127} = 2^{104}$ soit environ 2 × 10^{31}. La perte de précision à cet exposant est donc colossale.

Enfin, un tableau dans lequel nous placerons la durée des traitements est déclaré globalement afin de ne pas avoir à le passer en paramètre à chaque sous-programme. Ce type de programmation n'est pas très sûr, mais n'est ici utilisé que pour la phase de mise au point et ne sera pas présent dans le programme final.

```
// Tableau utilisé pour stocker les durées des parties du
// programme
// 0: lecture analogique de température et calculs
// 1: lecture TPM
// 2: envoi sur le port série
unsigned long durees[3];
```

2.2 Lecture analogique de température

La température mesurée sera donnée en volts, sous forme d'un entier, par la fonction de l'API Arduino `analogRead()`. Il est assez rare, dans une API, que la lecture d'une entrée analogique renvoie un entier, et de nombreuses API renvoient le voltage lu sous forme d'un flottant. Malgré les apparences, le renvoi sous forme d'entier est, à notre sens, souhaitable, car la résolution du convertisseur analogique numérique est discrète, alors qu'un flottant peut entraîner une perte de précision. L'entier doit être interprété sous la forme d'un point fixe. Pour simplifier, un point fixe est un entier avec unité non unitaire. L'Arduino Uno possède un convertisseur dix bits, lié à une entrée [0-5 V]. Cela signifie qu'une lecture de 5 V correspondrait à la valeur 1 024, et que l'unité du point fixe renvoyé par `analogRead()` est le 1/1 024e de volt. Si l'on souhaite connaître la mesure en volts, il nous faut donc opérer `analogRead(A0)*5/1024.0`.

Convertisseur analogique numérique

Un convertisseur analogique-numérique (CAN) convertit un signal analogique (évoluant dans le domaine continu) en donnée numérique, donc discrète. Il est caractérisé notamment par sa résolution en bits. Ainsi, le CAN de notre Arduino Uno possède une résolution de dix bits, permettant de coder `N` = 2^{10} = 1 024 valeurs différentes. Par défaut, la tension de référence utilisée est [0-5 V], ce qui donne 5 V/1 024 unités = 4,88 mV/unité. Pour notre sonde TMP36, qui a une résolution `deltaV` = 10 mV/°C, cela signifie que, du fait de la discrétisation de la tension analogique d'entrée, deux mesures ont un écart minimal d'environ 0,5 °C. Notons que cela est inférieur à la précision du capteur, cependant cela peut être un problème dans une application où la variation de température, même fine, est importante. En effet, la précision absolue du capteur est supérieure à la résolution du CAN, cependant, on peut avoir un peu plus confiance en la variation du capteur

qu'en la lecture absolue de température, dans ce cas la résolution du CAN peut être un problème. Il est possible de changer la référence de mesure analogique, et l'Arduino Uno en possède une alternative qui est de [0-1,1 V]. Si l'on choisit cette référence, nous avons toujours 1 024 valeurs possibles, mais pour coder cette fois 1,1 V, ce qui correspond à 1,1 V/1 024 unités = 1,07 mV/unité. En choisissant cette référence, nous sommes capables de distinguer 0,1 °C d'écart en entrée. Cependant, cela restreint le domaine de fonctionnement de notre application : la sonde TMP36 fournit une tension comprise entre 0 V et 1,75 V pour une température variant de `Tmin` = –50 °C à 125 °C, par conséquent, comme nous ne pouvons alors lire que 1,1 V au maximum, nous ne pouvons lire les températures que jusqu'à 65 °C. Une solution alternative serait d'utiliser une tension externe en tension de référence : il est possible sur un microcontrôleur ATmega (type de celui qui équipe notre Arduino), d'utiliser une référence quelconque entre 0 V et 5 V pour la lecture analogique. Si nous utilisions une référence externe de 1,75 V, nous obtiendrions une résolution liée à la conversion analogique numérique correspondant à 0,17 °C, tout en conservant un domaine de fonctionnement [–50 °C à 125 °C].

Ainsi la figure 1.6 montre sur trois courbes les valeurs de 100 températures mesurées toutes les 50 ms avec les références de 5 V, 1,1 V et 1,75 V. Les mesures ont été effectuées en intérieur, avec une seconde d'écart entre les campagnes de mesures, qui duraient cinq secondes chacune. Par conséquent nous devrions lire des valeurs similaires. La conversion entre mesure de tension d'entrée analogique et température a été réalisée par l'opération : `((référence/N)*mesure)/deltaV + Tmin`.

Cependant, nous pouvons constater des différences absolues de températures. Cela n'est pas étonnant, la conversion étant très sensible à la référence, une petite erreur sur la tension de référence entraîne une différence de température calculée. Nous voyons donc qu'il est important de calibrer le capteur : si nous connaissons la température exacte, nous pouvons calculer plus exactement la tension de référence une fois pour toute. Le second point à observer est que les variations de température sont relativement précisément mesurées, et d'autant plus que la référence est faible, ce qui permet de profiter au maximum de la résolution. Ainsi, avec une référence de 5 V, la différence minimale entre deux températures est bien plus importante qu'avec la référence de 1,1 V.

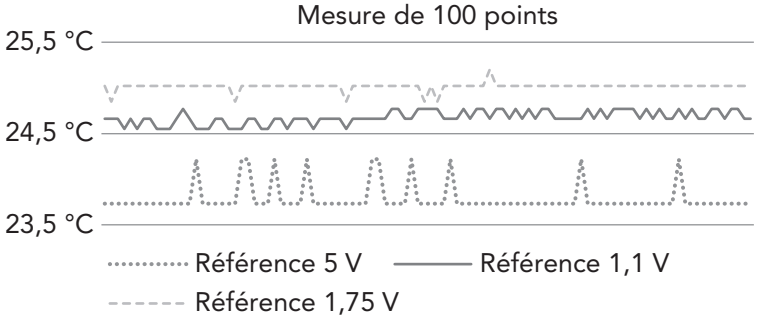

Figure 1.6 – Mesures de températures effectuées sur la TMP36 avec des références différentes.

Points fixes

La valeur renvoyée par `analogRead()` est typique de ce qu'on appelle un nombre à virgule fixe, ou point fixe. Ce nombre s'interprète comme un entier avec une unité qui n'est pas unitaire, ici, l'unité est le 5/1 024e de volts (avec la référence 5 V). Certains langages de programmation, comme Ada, proposent des éléments syntaxiques permettant au compilateur de vérifier que le développeur utilise ses points fixes de manière consistante. En C, le langage ne permet pas la définition de points fixes, et il est de la responsabilité du programmeur d'en assurer la consistance.

Types de données de base

La plupart des langages de programmation s'inspirent largement du langage C pour la taille des données. Cependant, la taille des types de base varie en fonction de l'architecture du processeur et potentiellement du système d'exploitation s'exécutant sur celui-ci. Les architectures de microprocesseur sont classifiées en catégories selon que les entiers de base `int` et les adresses (pointeurs) soient 16, 32 ou 64 bits. Ainsi on nomme ces architectures à base de quatre lettres « I » pour `int`, « L » pour `long`, « LL » pour `long long`, « S » pour `short`, et « P » pour pointeur. Une architecture LP64 propose des entiers longs (type `long`) et des pointeurs 64 bits, les entiers classiques `int` étant 32 bits.

Le tableau 1.1 donne la taille des entiers et pointeurs dans des architectures existantes. Par exemple, les architectures Microsoft Windows 64 bits sont généralement LLP64, alors que les architectures Unix 64 bits sont généralement ILP64. À l'exception du type `char`, la taille des types varie d'une architecture à une autre, il est bon de ne pas faire d'hypothèse sur la taille de représentation d'un type de base, et l'opérateur `sizeof()` doit toujours être utilisé lorsqu'on a besoin de la taille d'un type ou d'une variable scalaire.

Remarque

`char` signifie caractère, car historiquement, le code ASCII encodant les caractères américains était exprimé sur sept bits, auquel on adjoignait un bit tel que chaque octet ait une parité paire, ce qui permettait de détecter des erreurs de transmission entre le clavier et l'unité centrale des premiers ordinateurs. Ce code a ensuite été étendu à huit bits, ajoutant ainsi 128 caractères et symboles possibles. L'encodage des caractères est un défi, et la famille de normes Unicode s'y intéresse, avec des caractères pouvant être codés sur un à quatre octets (en UTF-8, par exemple).

Tableau 1.1 Taille des entiers et pointeurs dans des architectures classiques.

Modèle d'architecture	`char`	`short`	`int`	`long`	`long long`	pointeur
LLP64	8 bits	16 bits	32 bits	32 bits	64 bits	64 bits
LP64	8 bits	16 bits	32 bits	64 bits	64 bits	64 bits
ILP64	8 bits	16 bits	64 bits	64 bits	64 bits	64 bits
ILP32	8 bits	16 bits	32 bits	32 bits	32 bits	32 bits
LP32	8 bits	16 bits	16 bits	32 bits	32 bits	32 bits
SILP64	8 bits	64 bits	64 bits	64 bits	64 bits	64 bits