Complément d'information, **INF1900**

École Polytechnique de Montréal

Version 1.4 (16 septembre 2020)

par Philippe Proulx

Historique des versions

Date	Version	Détails
20 octobre 2010	1.0	• Document initial
30 octobre 2010	1.1	• Corrections de français écrit • Modifications d'exemples et de schémas • Numéros de pages comptent à partir de la première page • Corrections après première révision par Jérôme Collin
27 août 2012	1.2	• ATmega16 → ATmega324PA • Quelques améliorations mineures
22 mai 2018	1.3	• Ajustements pour inf1900 par Jérôme Collin
16 septembre 2019	1.4	• Nouveau lien vers le site du cours

Introduction

Ce document non officiel constitue un complément d'information pour le cours INF1900 de l'École Polytechnique de Montréal. Son mandat n'est donc pas de remplacer la théorie enseignée dans le cours, mais bien d'éclairer le lecteur sur certains points et ainsi d'être un supplément. Une excellente ressource demeure le site Web du cours, <http://cours.polymtl.ca/inf1900/>, où se trouvent également certains sujets abordés ici.

Après deux sessions en tant que répétiteur du cours, j'ai remarqué que, bien que la théorie semble assez bien maitrisée par la plupart des étudiants, une grande partie d'entre eux ont de la difficulté à s'orienter et à trouver des ressources au début. Entre autres, plusieurs trouvent les documentations officielles écrasantes et ne prennent pas assez la peine de les feuilleter faces à un problème. Le but du document est donc :

- de montrer au lecteur **comment lire les deux documentations importantes** pour le cours (celle de la librairie AVR Libc, <http://www.nongnu.org/avr-libc/user-manual/>, et celle du microcontrôleur ATmega324PA;
- de **bien expliciter** la structure, l'architecture et le concept du microcontrôleur utilisé;
- de décrire le **processus de compilation** et de programmation (dans le sens « téléchargement d'un programme assemblé vers le microcontrôleur » du terme) du microcontrôleur et
- d'apporter **différentes précisions** quant à certains points spécifiques et pratiques de la librairie AVR Libc.

Comme il est possible de le constater, la matière fondamentale du cours n'est pas couverte par ce document (à titre d'exemple, les notions de PWM et de machine à états ne sont pas expliquées); il se veut simplement un guide relativement complet qui permet au lecteur d'éviter une perte de temps accrue sur une kyrielle de détails.

Un danger subsiste néanmoins : ce document ne devrait pas être lu avec comme seul objectif l'achat facile de temps; bien comprendre les concepts explorés ici permet d'élaborer un meilleur design pour la réalisation des travaux pratiques et du projet final du cours, en plus de rendre le lecteur plus à l'aise avec les outils utilisés et d'attiser sa « maturité informatique » en cours de développement.

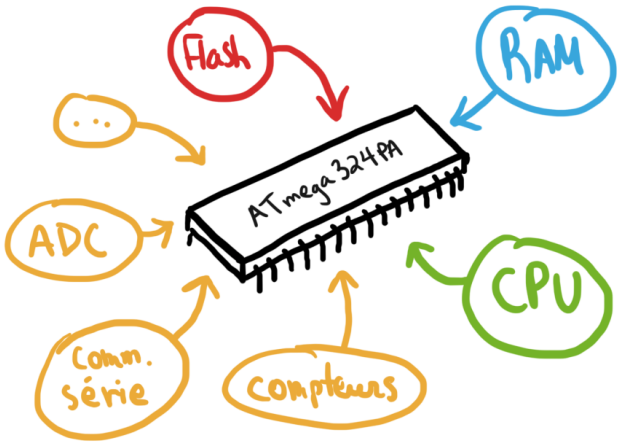
Bien que ce document puisse servir de référence après une première lecture, je vous conseille de le lire en ordre au début afin de vous rendre plus confortable avec l'ensemble des notions distinctes. Le texte couvre d'abord l'architecture du microcontrôleur utilisé, l'AVR ATmega324PA, de façon plus informelle et simple que sa documentation officielle. Une fois cette base acquise, la lecture de la documentation officielle de Microchip est abordée. Ces deux sections permettent au lecteur de comprendre « ce qui se passe », mais elles ne sont pas suffisantes pour être en mesure de réaliser le moindre projet, ce pourquoi les outils utilisés, la compilation et la librairie AVR Libc sont abordés par la suite.

Le microcontrôleur : l'AVR ATmega324PA

Le défi du cours INF1900 est de programmer un microcontrôleur afin de faire interagir un logiciel avec le monde réel par le biais de capteurs et d'autres entrées/sorties. Contrairement aux cours plus orientés autour du logiciel, donc, où la partie matérielle n'est pas considérée dans le problème, l'étudiant (« vous », à partir de maintenant) d'INF1900 doit produire et lire des signaux électriques aux bornes des pattes du microcontrôleur utilisé… toute une nouveauté! Étant donné que le cours au complet est passé à programmer cette puce, mieux vaut bien la comprendre de façon générale afin de mieux pouvoir aborder une de ses fonctions spécifiques au moment voulu. Cette section présente l'architecture de l'ATmega324PA. Mais d'abord, quelques réponses importantes.

Qu'est-ce qu'un microcontrôleur?

Un microcontrôleur (souvent abrégé « µC » ou « MCU ») est réellement un petit ordinateur dans une boite fermée. C'est environ l'équivalent, si on veut, de la carte mère d'un PC en plus petit et avec moins de ressources : la boite fermée contient un microprocesseur, de la mémoire Flash (donc persistante lors d'absence de courant électrique), de la RAM, des périphériques externes au microprocesseur (mais pas externes à la boite fermée) et des ports d'entrée/sortie.



Pourquoi utiliser un microcontrôleur dans la conception d'un robot?

Tel que décrit plus haut, un microcontrôleur contient l'ensemble des composants nécessaires à la mise en place d'un ordinateur, et à relativement faible coût dans notre contexte. Nous aurions également pu utiliser un simple microprocesseur, tel qu'un vieux Pentium, mais nous aurions alors dû l'entourer de mémoire Flash, de RAM et de certains périphériques nécessaires au projet. Tous ces ajouts augmentent la complexité du circuit et rendent la vie beaucoup plus difficile à l'étudiant pour un coût qui, au final, n'est probablement pas plus bas que celui d'un microcontrôleur. De plus, la documentation du microcontrôleur utilisée est complète et de bonne qualité; ceci évite d'avoir plusieurs documents différents à consulter.

Le secret de la réussite : les mémoires du microcontrôleur

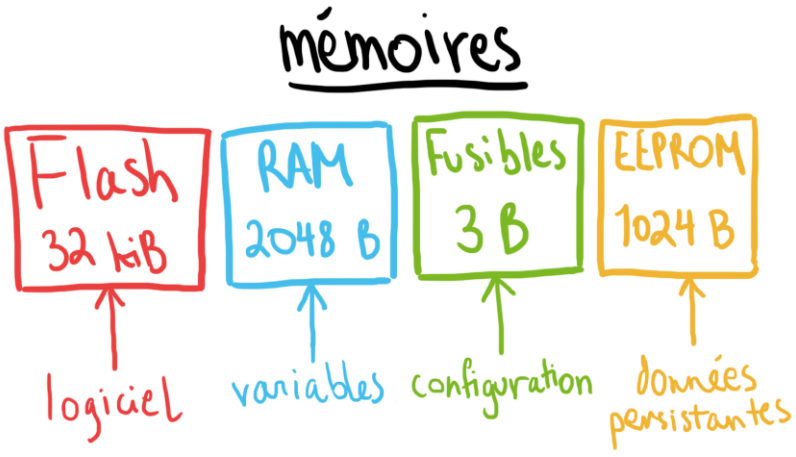
Ce n'est pas nécessairement un sujet que l'on désire aborder au début, les mémoires, lorsque notre seul but est de faire clignoter une DEL, mais il s'agit réellement du cœur du microcontrôleur. Rapidement, vous comprendrez que la seule façon de contrôler tous les aspects de l'ATmega324PA est de jouer avec le contenu de ses mémoires.

L'AVR ATmega324PA contient **quatre mémoires** différentes importantes pour le contexte. La première est de trois octets seulement et contient les ***fusibles***. Les fusibles sont trois octets de configuration du microcontrôleur qui ne changent pas souvent (tel qu'on ne change pas souvent des fusibles en général). Les configurations en question ont souvent rapport à l'environnement de la puce, tel que l'emplacement de l'horloge (externe ou interne). Dans le cadre d'INF1900, ces valeurs sont fixées une seule et unique fois, au premier ou au deuxième cours, lorsque vous venez faire « initialiser » vos microcontrôleurs par les chargés et répétiteurs.

La seconde est une **mémoire Flash**. Cette mémoire est persistante et donc ne s'efface pas en absence de courant électrique. Elle contient les instructions assemblées sous forme de code machine de votre logiciel. Sa taille de 32 kiB justifie le « 32 » de « ATmega324PA »; vous devinez donc qu'un ATmega8 offre 8 kiB de mémoire Flash. Le processus d'écriture dans cette mémoire est assez lourd, et est réalisé par un *programmeur*, un outil qui écrit le code machine de votre logiciel dans cette mémoire (ce processus est appelé la *programmation* de la puce). On parle donc ici d'une mémoire en lecture seule par votre logiciel.

Une troisième mémoire est de **type EEPROM** (*Electrically Erasable Programmable Read-Only Memory*), comprend 1024 octets et sert d'espace de stockage permanent pour les données que votre logiciel veut bien y stocker (voyez-la comme un minuscule disque dur). Ceci dit, elle n'est pas utilisée en INF1900, pour des raisons d'ailleurs mystérieuses.

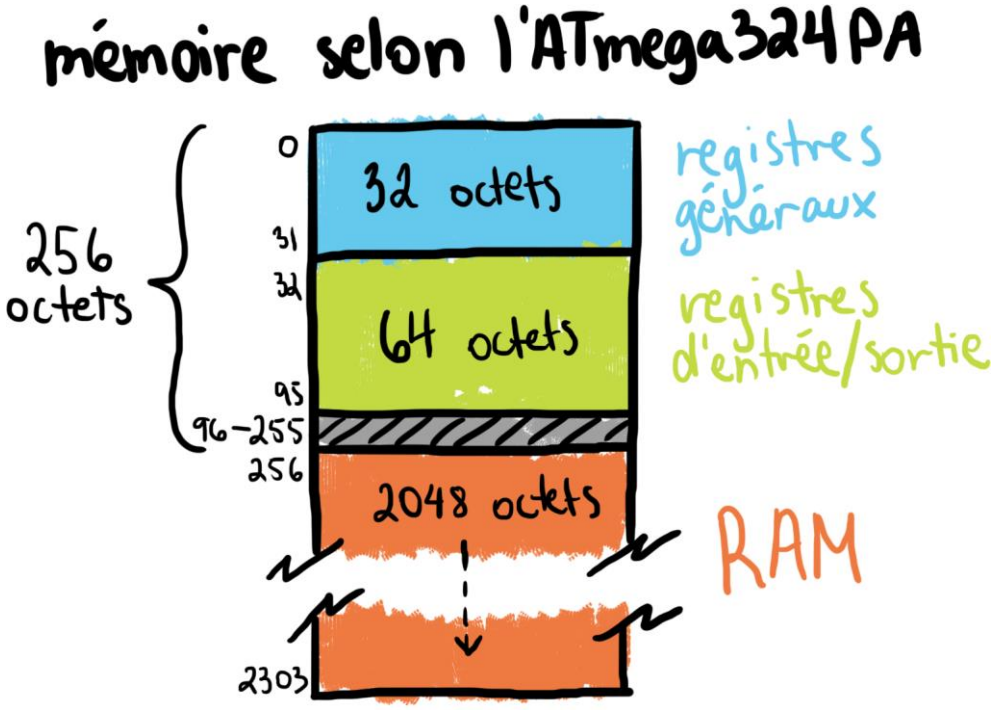
Enfin, on retrouve 2 kiB de **RAM** (*Random-Access Memory*). Cette mémoire très rapide permet de stocker des variables et d'autres informations temporaires (comme dans « temporaires à la présence d'alimentation électrique »). Dès que le courant électrique coupé, ou peu après, elle perd ses données. Je suis assez convaincu que vous avez déjà entendu l'acronyme RAM à votre étape; sur votre PC, vous en avez peut-être pour 2 GiB ou 4 GiB, respectivement 1 048 576 et 2 097 152 fois plus que l'ATmega324PA. Vous serez toutefois surpris de ce qu'il est possible de réaliser avec aussi peu!



Attention! La « mémoire externe » sur la carte mère du projet **ne fait pas partie du microcontrôleur**. Il s'agit d'une autre puce à laquelle il est possible d'accéder grâce au microcontrôleur, mais qui est bel et bien externe à celui-ci.

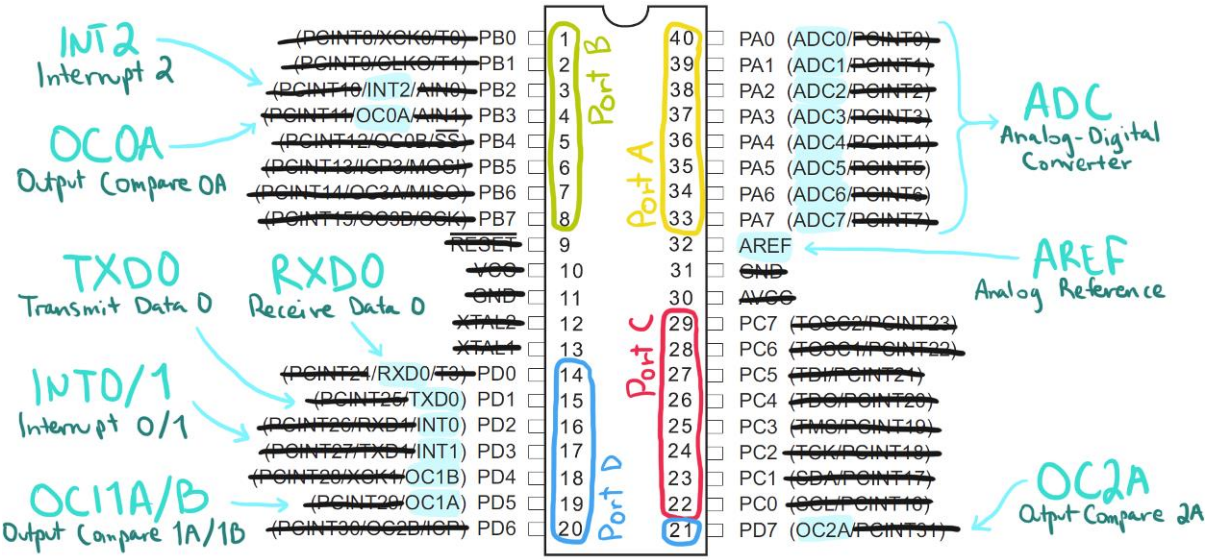
La partie de ce document qui vous demande probablement le plus de concentration commence ici. Comme tout microprocesseur, celui de l'ATmega324PA contient différents registres. Un registre est une **petite mémoire très rapide** et temporaire. Il peut servir à stocker des données quelconques pendant un moment, ou à contrôler certaines fonctions du microcontrôleur. Si vous suivez le cours INF1600 (Architecture des micro-ordinateurs) en même temps, comme c'est souvent le cas, vous comprendrez bien rapidement ce que constitue un registre. Pour l'instant, dites-vous que c'est une petite mémoire, de 8 bits (1 octet) dans le cas de l'ATmega324PA. Ce microcontrôleur contient exactement 256 registres. Parmi ceux-ci, 32 sont dits « généraux », c'est-à-dire qu'ils peuvent être utilisés pour stocker n'importe quoi de pertinent à votre logiciel durant l'exécution. Les autres sont des registres de contrôle, de statut et de données relatifs aux fonctions du microcontrôleur.

Ce qui est particulier, c'est que le microcontrôleur considère ces registres comme faisant globalement partie de sa « mémoire ». Ainsi, il attribue une adresse à chacun de ceux-ci. Les adresses 0 à 31 contiennent les 32 registres généraux et les adresses 32 à 95 les 64 autres registres (dits d'entrée/sortie). On retrouve également sur l'ATmega324PA 160 registres dits « extended I/O » que nous n'utilisons pas en INF1900. Le total, 32 + 64 + 160, équivaut bien à 256. À partir de l'adresse 256, on accède à la RAM (donc de 256 à 2303 inclusivement pour former les 2048 octets de RAM disponibles). Heureusement, les outils que nous utilisons pour programmer dans le cours permettent de faire complètement abstraction de l'adressage. Il est quand même important de comprendre le concept afin de mieux déchiffrer la documentation de Microchip et celle d'AVR Libc, qui seront vues plus loin dans ce document.



Brochage de l'ATmega324PA

Le *brochage* (« *pinout* » en anglais) d'une puce décrit le rôle de chaque broche de celle-ci. Pour l'ATmega324PA, on retrouve souvent le schéma classique suivant, sur lequel je me suis permis certaines notes.



Le schéma typique du brochage contient donc normalement le nom de la broche et entre parenthèses, un ou plusieurs **rôles alternatifs** que peut prendre la broche, à la discrétion du développeur du logiciel (en l'occurrence, vous). Ici, les rôles alternatifs qui ne seront pas utilisés en INF1900 sont raturés. Vous pouvez toujours les consulter par curiosité, mais ce n'est pas obligatoire. Sont aussi raturés les noms des broches avec lesquelles votre logiciel n'interagira pas directement. Pour le reste, les acronymes et abréviations des rôles alternatifs sont décrits (inutile de les comprendre pour l'instant, vous aurez toute la session à cette fin).

Notez qu'on distingue **quatre ports** : A, B, C et D. Un port est ici un ensemble de **huit broches** précises. Lorsque ces ports ont leurs rôles généraux (et non leurs rôles alternatifs), vous avez sur eux un contrôle total. Vous pouvez donc assigner à n'importe quelle de ces 32 broches une valeur logique. Si une broche est « vraie », le microcontrôleur applique environ 5 V par rapport à la masse sur celle-ci (cette tension dépend de l'alimentation de la puce, mais il est inutile d'examiner les propriétés électriques dans ce document). Si une broche est « fausse », alors le microcontrôleur met en sortie la masse (donc 0 V par rapport à la masse). Mais une broche peut aussi « lire » une valeur logique : elle lira « vraie » la broche si on lui applique de façon externe une tension de 5 V par rapport à la masse et « fausse » si on lui applique la masse. En tout temps, une broche est soit en mode lecture, soit en mode écriture; le mode, ou la *direction* des données, est décidé par le logiciel, et donc ultimement par vous.

Comment écrire une valeur logique sur une broche? Il se trouve que ces quatre ports prennent les valeurs de quatre registres spécifiques d'entrée/sortie. Les broches prennent alors la valeur des bits correspondant dans le registre concerné. Par exemple, écrire la valeur binaire 01101101 (binaire) dans le registre du port C fera en sorte que les broches PC6, PC5, PC3, PC2 et PC0 présenteront 5 V par rapport à la masse et le reste des broches du port C la masse. Ce contrôle sur la tension des broches individuelles permet une interaction avec le monde réel; on

convertit ici une valeur logique (« vrai » ou « faux ») en valeur physique réelle (une tension).

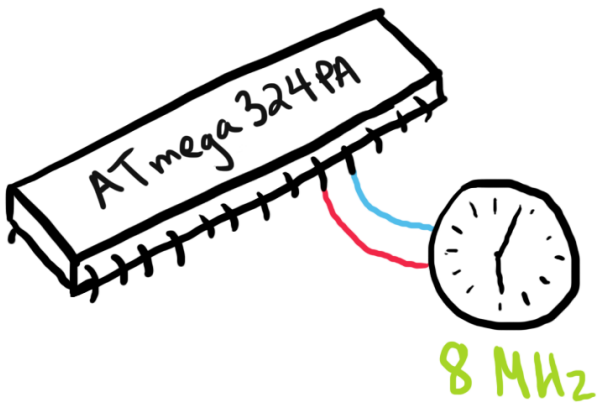
Pour des raisons de commodité, le circuit imprimé monté que constitue la petite carte mère du projet présente quatre connecteurs IDC mâles à 10 broches chaque. Vous devinez qu'il s'agit des quatre ports qui sont reliés, par le circuit, aux broches en question. Sur chacun de ces connecteurs, une broche est à 5 V par rapport à la masse et l'autre est la masse. Il en reste donc 8, respectivement pour les 8 broches du port. À noter que, sur la carte mère, les identificateurs imprimés aux côtés des broches des connecteurs IDC commencent à compter à partir de 1. Les broches sont donc identifiées de 1 à 8, alors que dans la documentation de Microchip, **on compte de 0 à 7**. Cette dernière convention est en fait plus juste puisque la broche 0 correspond au bit 0 du registre représentant le port, et non au bit 1. Il y a donc une concordance intéressante entre l'indice des bits du port et l'indice des broches. Vous devriez toujours adopter la convention de Microchip.

Horloge

L'horloge (« *clock* ») du microcontrôleur est ce qui dicte la vitesse des instructions lues de la mémoire Flash par le microprocesseur. Vous n'êtes probablement pas en mesure de comprendre la notion d'*instruction* à cette étape, mais sachez qu'il s'agit d'une opération qui effectue un transfert entre des registres ou un calcul. À la longue, l'ensemble de ces instructions exécutées rapidement forment votre logiciel. Nous n'aurons pas, en INF1900, à programmer directement ces instructions. Effectivement, nous utiliserons un langage de programmation de plus haut niveau qui est « traduit » en instructions par un processus automatisé. La plupart des instructions se font en un seul cycle d'horloge.

L'ATmega324PA possède une horloge interne qui se situe à environ 8 MHz, selon l'alimentation et la température. Il offre également la possibilité de fournir une horloge plus juste par le biais d'un circuit externe. La vitesse maximum supportée est de 20 MHz. Dans le cadre d'INF1900, nous fournissons un circuit externe qui procure une horloge de 8 MHz. C'est donc environ 8 000 000 instructions par seconde qui sont exécutées par le microprocesseur de l'ATmega324PA!

Une petite révision des principales équations reliées à la fréquence ne fera pas de tort pour INF1900. Par exemple, savoir passer facilement d'une fréquence à une période, ou encore calculer le nombre de cycle d'horloge nécessaire, connaissant la fréquence, pour atteindre une durée donnée.



La documentation officielle de l'ATmega324PA

Plus que vous le croyez, vous aurez à fouiller dans la documentation officielle de l'ATmega324PA, écrite par les ingénieurs et rédacteurs techniques de Microchip, afin de comprendre comment contrôler différentes fonctions du microcontrôleur. Cette section explique comment lire cette documentation officielle et où chercher des solutions aux problèmes rencontrés.

Sections générales

Sachez d'abord que les signets du document PDF aident énormément. Vous devriez toujours naviguer dans cette documentation à l'aide des signets.

La documentation de l'ATmega324PA débute et se termine par quelques sections générales. Par « section générale », j'entends une section décrivant des concepts qui s'applique au microcontrôleur en entier et qui ne décrit pas une fonction particulière. Dans le cas d'une maison, il s'agirait de décrire la toiture, le nombre de pièces, le prix ou la superficie du terrain plutôt que de présenter chaque pièce individuellement, la piscine ou l'intérieur des cabanons. J'espère que l'analogie est appropriée. Les sections générales sont tout ce qui concerne :

- le microprocesseur (« AVR CPU Core »);
- les mémoires (« AVR memories »);
- l'horloge (« System clock and clock options »);
- la gestion de l'énergie (« Power management and sleep modes »);
- le contrôle du système (« System control and reset »);
- les interruptions en général (« Interrupts »);
- les ports d'entrée/sortie (« I/O-Ports »);
- les sections concernant la programmation et le débogage (« JTAG interface and on-chip debug system », « IEEE 1149.1 (JTAG) Boundary-scan », « Boot loader support – read-while-write self-programming » et « Memory programming »);
- les caractéristiques électriques et typiques (« Electrical characteristics » et « Typical characteristics ») et
- les informations sur le boitier (« Packaging information »).

En principe, vous n'avez pas à lire ces sections. Les introductions sur la mémoire et sur l'horloge faites dans ce document sont suffisantes pour le cours. Je vous invite néanmoins à les consulter pour plus de détails sur le fonctionnement du sujet concerné par chacune de ces sections, si vous avez le temps.

Fonctions spécifiques du microcontrôleur

L'ATmega324PA fournit certaines fonctions intéressantes dans le cadre d'un système embarqué. Les sections de la documentation qui ne sont pas générales sont habituellement des sections décrivant chacune une fonction bien particulière.

Parmi celles-ci, voici celles dont la lecture n'est pas directement nécessaire à INF1900 au début :

- l'interface du protocole SPI (« SPI – Serial Peripheral Interface » et « USART in SPI mode »);
- l'interface du protocole I²C (« Two-wire Serial Interface »);
- le convertisseur analogique-digital (« Analog-to-digital converter ») et
- le comparateur analogique (« Analog Comparator »).

Ceci nous laisse avec les sections suivantes, qui devront être abusivement lues pour une bonne réussite du cours :

- les **interruptions externes** (« External Interrupts »);
- les **quatre compteurs** (« 8-bit Timer/Counter0 with PWM », « 16-bit Timer/Counter1 and Timer/Counter3 with PWM » et « 8-bit Timer/Counter2 with PWM and asynchronous operation ») et
- la **communication série universelle** (« USART »).

Il est à noter que certaines fonctions dont la section n'est pas nécessairement à lire seront quand même utilisées. Seulement, du code source complet (habituellement, des classes C++) sera fourni par les énoncés des travaux pratiques, ce qui vous évitera d'avoir à lire la documentation officielle.

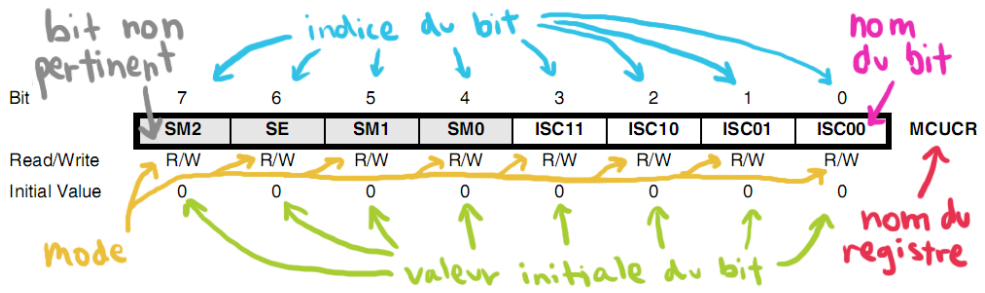
Principe des fonctions spécifiques du microcontrôleur

À force de jouer avec celles-ci un peu, on découvre que le processus pour utiliser les fonctions spécifiques est toujours le même. Voici ce qu'il y a à savoir :

- une fonction spécifique fera habituellement en sorte qu'une ou plusieurs broches d'un des quatre ports prendra son **rôle alternatif**;
- afin d'activer et de configurer une fonction spécifique, il suffit d'écrire des valeurs bien spécifiques, décrites en détails dans la documentation officielle, dans des **registres d'entrée/sortie bien précis** et
- afin d'obtenir le statut ou une donnée d'une fonction spécifique, il suffit de lire de **registres d'entrée/sortie bien précis**, décrits en détails dans la documentation officielle.

Le « gros de la job » lorsque vous désirez utiliser une fonction du microcontrôleur est donc de lire sa description et son fonctionnement (les premiers paragraphes de sa section), de prendre connaissance des paramètres disponibles pour celle-ci et de les ajuster grâce aux registres d'entrée/sortie concernés.

Les registres concernés et le rôle de chaque bit de ceux-ci sont toujours représentés par un tableau horizontal. L'exemple suivant montre le registre MCUCR de l'ATmega324PA (le microcontrôleur que nous utilisions lors des sessions précédentes), le « MCU Control Register », avec l'ajout personnel d'indications.



Chaque bit :

- possède un **indice** (de 0 à 7, parfois de 0 à 15 lorsqu'il s'agit d'un registre de 16 bits, qui n'est en fait que la concaténation de deux registres de 8 bits);
- possède un **nom symbolique**;
- possède un **mode** (détermine si le bit est destiné à être toujours lu (« R »), toujours écrit (« W ») ou les deux (« R/W »));
- possède une **valeur par défaut** (il s'agit de la valeur de ce bit pour le registre concerné lors de l'initialisation du microcontrôleur, c'est-à-dire suite à une remise à zéro, ou un *reset*) et
- est **pertinent ou non** au contexte de la fonction décrite dans la section lue (tel qu'énoncé précédemment, certains registres sont partagés par plusieurs fonctions; ainsi, un bit possédant une cellule grise dans le tableau décrivant le registre n'est pas à considérer pour la fonction décrite).

Il faut faire attention au dernier point : bien qu'un bit ne soit pas pertinent à la fonction décrite, il ne faut pas le modifier, car ceci pourrait influencer une autre fonction. Il est donc important de prendre les mesures nécessaires pour laisser les bits non pertinents tels quels.

Il faut également faire attention de bien lire la description de chaque bit. Autrement dit, ce n'est pas parce qu'un bit ne vous concerne pas que vous devriez sauter la lecture de sa description, puisque vous jouez avec le registre le portant. À titre d'exemple, il arrive parfois qu'un certain bit doive être « vrai » au cours de l'écriture d'une valeur dans un registre donné, sans quoi l'écriture ne se fera pas. La stratégie générale est donc :

1. lire la description **de chaque bit pertinent** du registre à modifier;
2. **déterminer sa valeur**, « vraie » ou « fausse », selon le problème et
3. trouver comment assigner les nouvelles valeurs seulement aux bits pertinents afin de **ne pas affecter** les bits du registre appartenant à une autre fonction.

Quelquefois, les premiers paragraphes d'une section décrivant une fonction spécifique peuvent être lourds à lire. Les auteurs assument qu'un ingénieur muni d'un minimum d'expérience avec les microcontrôleurs est le lecteur typique de la documentation officielle. Je conseille alors d'aller lire sur le sujet à l'extérieur de cette documentation, en particulier sur Wikipedia (version en anglais, <http://en.wikipedia.org/>) où les concepts sont souvent bien vulgarisés.

AVR Libc – l'indispensable librairie

Une *librairie* (ou *bibliothèque*), en génie logiciel, est une collection de routines et possiblement de classes venant en aide au développement logiciel. Tout comme une bibliothèque réelle, une librairie contient des procédures réutilisables et partagées (par analogie aux livres).

Dans le contexte d'INF1900, la librairie AVR Libc regroupe :

- des **définitions d'entêtes** correspondant aux symboles de la documentation officielle de Microchip;
- des fonctions analogues à celles de la **librairie standard C** et
- des **fonctions utilitaires** spécifiques à la conception de logiciels pour un microcontrôleur d'architecture AVR.

Cette librairie est écrite en langage C et rend ses fonctions disponibles en langage C (ou C++, étant donné la compatibilité implicite entre les deux langages). Dans le cadre du cours, les principaux avantages sont :

- la possibilité d'écrire un logiciel pour l'ATmega324PA **en langage C ou C++**, nous évitant ainsi d'écrire directement de l'assembleur (« ASM ») AVR;
- l'**abstraction de l'adressage** de la mémoire grâce aux définitions d'entêtes (fichiers habituellement inclus dans les sources C/C++ avec la directive `#include`) et
- la **disponibilité de fonctions utilitaires** pratiques telles que celles concernant les délais.

Le second point est particulièrement intéressant. Nous nous souvenons que, par exemple, le port B de l'ATmega324PA correspond en fait à un registre d'entrée/sortie et qu'attribuer une valeur à ce registre a pour effet de modifier les tensions aux bornes des broches du port. En fouillant dans la documentation officielle de Microchip, on trouve dans la section « Register summary » que le port B se trouve à l'adresse 0x25 (c'est-à-dire 37) en mémoire. Grâce à AVR Libc, ce registre devient une variable globale, `PORTB`, et la modification de cette variable affecte le registre à l'adresse 37. Il est donc tout à fait possible de programmer un logiciel complet sans ne jamais avoir recours aux adresses absolues des registres, qui pourtant existent.

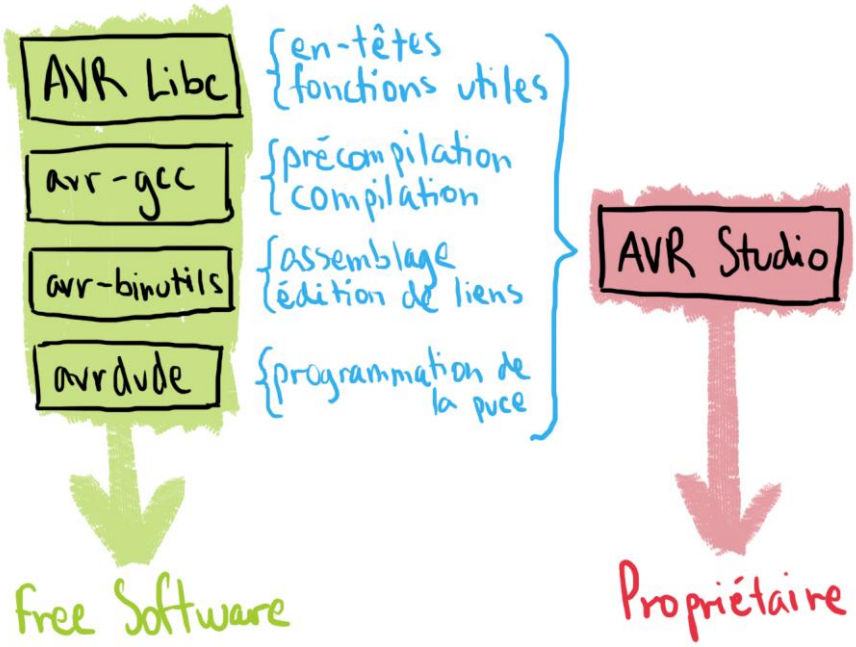
Les auteurs d'AVR Libc ont fait attention de choisir les symboles des variables et des définitions en fonction de ceux de la documentation officielle de Microchip. Ainsi, la modification du registre MCUCR correspond à la modification de la variable globale `MCUCR` en langage C.

Différence entre les termes

Il est important de bien comprendre la différence entre les termes désignant les outils mis à notre disposition. Les trois principaux sont **AVR Libc**, **avr-gcc** ainsi qu'**avr-binutils**. AVR Libc, décrite au début de la section, ne permet pas de compiler votre code source en langage C vers un fichier à télécharger vers la mémoire Flash de l'ATmega324PA; avr-gcc et avr-binutils sont plutôt les deux suites d'outils qui permettent ce processus. Donc, en soi, vous ne pouvez rien faire avec AVR Libc si

vous n'avez pas également les deux autres. Évidemment, tous les PC des laboratoires utilisés en INF1900 sont équipés de tout ce qu'il faut.

Tel qu'énoncé sur la page d'accueil d'AVR Libc, les composantes avr-gcc, avr-binutils et AVR Libc forment la solution Free Software (voir la philosophie : <http://www.gnu.org/philosophy/free-sw.html>) de développement pour la plupart des microcontrôleurs AVR. De son côté, Microchip fournit également une librairie et une série d'outils de compilation, AVR Studio, mais qui ne respecte pas les contraintes imposées par les licences du Free Software. L'image suivante résume les termes importants.



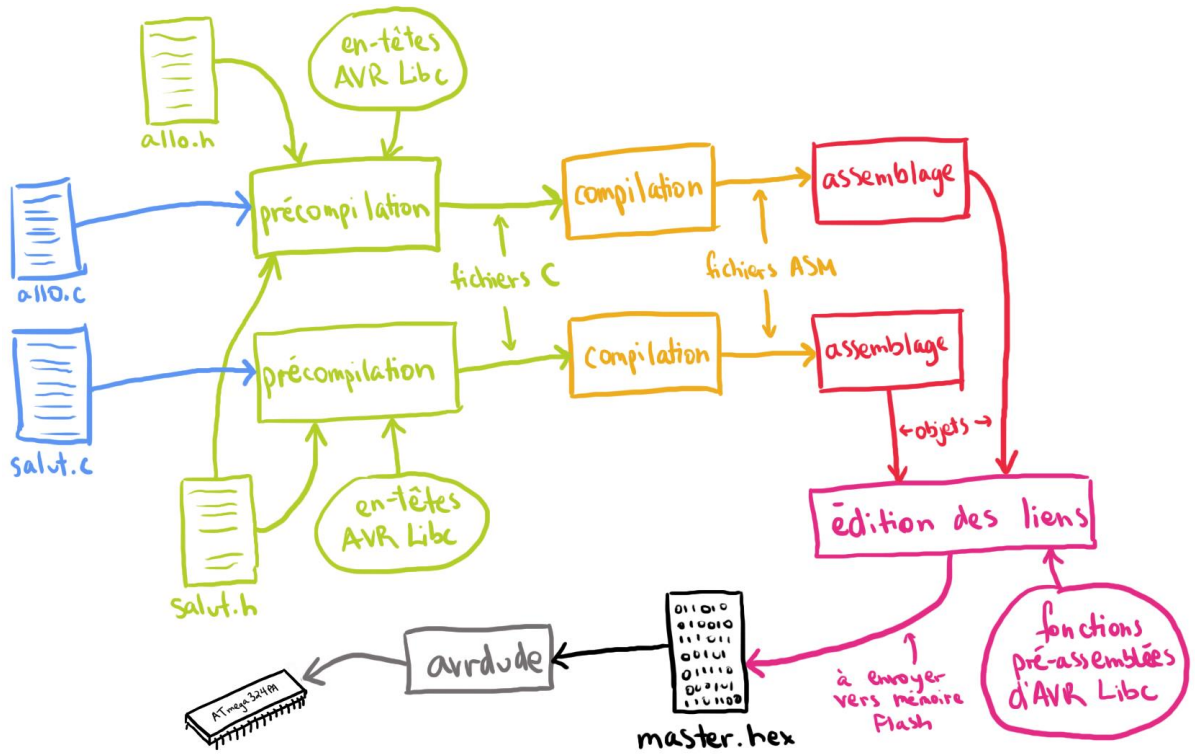
On remarque aussi ici avrdude, qui est l'outil permettant l'envoi du fichier de code machine (votre logiciel assemblé et lié) vers l'ATmega324PA.

Ici, avr-gcc est un port de gcc et avr-binutils un port de binutils, deux suites de GNU très populaires. La beauté de la chose réside dans le fait que ces outils produisent des fichiers destinés à une autre plateforme que celle sur laquelle ils sont exécutés. En effet, l'assembleur d'avr-binutils crée du code machine pour un jeu d'instructions AVR tandis qu'il est exécuté sur un processeur muni d'un jeu d'instructions IA-32 ou x86-64 (je le mentionne à titre d'information, mais attendez un peu plus tard en INF1600 afin de bien comprendre la signification de cette phrase). On qualifie alors avr-gcc ici comme étant un « *cross-compiler* ».

AVR Libc, quant à elle, est une version modifiée de libc, la librairie standard C. La librairie tente de respecter le standard le plus possible, tout en y ajoutant différents artefacts propres aux microcontrôleurs AVR.

Chaine d'étapes du processus de compilation

Le processus complet exécuté lorsque vous tapez votre fameux « `make install` » peut être divisé en plusieurs étapes plus simples. Chacune de celles-ci fait appel à une commande différente des suites introduites à la sous-section précédente. Loin de moi l'idée de vous faire apprendre par cœur la chaine d'étapes en question, mais un aperçu schématique simplifié permet de mieux situer où interagissent les différents outils et ainsi de mieux saisir le rôle de chacun.



Les étapes les plus importantes sont donc illustrées ici.

- La **précompilation** permet de créer des fichiers sources C complets et sans directives de précompilation (habituellement précédées du symbole `#` dans un fichier source); ainsi, les inclusions (`#include`) sont réalisées et les définitions (`#define`) partout dans le code sont développées. Le compilateur ne comprend pas les directives de précompilation; elles ne servent qu'à accélérer le travail du développeur (vous), ce qui signifie que vous pourriez effectivement vous en passer en incluant vous-même le contenu d'autres fichiers et en écrivant les valeurs des définitions directement partout dans les sources.
- La **compilation** prend un seul et unique fichier C et, grâce à un processus qui relève probablement de la magie, renvoie un fichier contenant la liste des instructions AVR qui font ce que l'on a décrit avec un plus haut niveau d'abstraction en langage C.
- L'**assemblage** produit des fichiers objets (attention, il ne s'agit pas de la même sémantique que les *objets* du langage C++), c'est-à-dire un fichier de code machine, grâce à une liste d'instructions. Cette conversion est environ un processus un-pour-un : les instructions sont déjà écrites dans le bon ordre, il ne reste qu'à interpréter le texte et à en faire un ou plusieurs octets pertinents.
- L'**édition des liens** est assez complexe et permet de prendre tous les fichiers objets, incluant potentiellement ceux d'une librairie déjà assemblée, et d'en faire un seul et unique fichier exécutable de sortie. Habituellement, une seule fonction ou un groupe de fonctions pertinentes se retrouve dans un fichier C, qui plus tard correspond à un fichier objet. Si une fonction d'un autre fichier est utilisée, le compilateur assume qu'elle existe et se ferme les yeux sur

l'histoire. Plus tard, le rôle de l'éditeur de liens est de rassembler tous ces fichiers objets et de s'assurer que tout le monde est là.

- Enfin, la **programmation du microcontrôleur** est effectuée par avrdude; il envoie alors le fichier sorti par l'éditeur de liens directement vers la mémoire Flash de l'ATmega324PA.

On voit donc qu'AVR Libc interagit à l'étape de précompilation et d'édition des liens. Ses entêtes sont inclus dans un fichier source à l'aide d'une directive de précompilation telle que « `#include <avr/io.h>` ». Mais où est `<avr/io.h>`? Les chevrons indiquent ici au précompilateur de regarder à la racine d'un de ses chemins par défaut. Ceci dépend de la configuration des outils de compilation. Sur les PC du laboratoire d'INF1900, un de ces chemins est `/usr/avr/include`. On trouve effectivement à partir de ce répertoire le fichier `avr/io.h`, qui lui-même inclut d'autres fichiers et ainsi de suite.

La liste de tous les entêtes d'AVR Libc se trouve à l'URL <http://www.nongnu.org/avr-libc/user-manual/modules.html>. On y aperçoit plusieurs noms connus originaux de libc : `<math.h>`, `<string.h>` et `<stdlib.h>` en sont quelques uns.

La documentation officielle d'AVR Libc

Lorsque vous vous questionnez sur une fonction ou une définition d'AVR Libc, il ne faut pas hésiter à aller consulter sa description textuelle en ligne. La page listant tous les entêtes (ou *modules*) est un bon point de départ. Ceci dit, il peut arriver que vous ne sachiez pas à quel entête appartient un symbole donné; la liste alphabétique des symboles, <http://www.nongnu.org/avr-libc/user-manual/globals.html>, est alors très appropriée.

Même si vous pensez bien connaitre le but d'une fonction ou d'une définition, vous devriez avoir comme rituel d'aller lire sa description quand même. Celle-ci contient souvent des précisions importantes et occasionnellement non négligeables dans le contexte de certains problèmes en INF1900. Encore une fois, si INF1900 est un cours assez pratique, il n'en demeure pas moins qu'il est aussi beaucoup orienté vers le développement de réflexes autodidactes (lire : lecture de documentations), nécessaires d'après moi à l'éducation de tout ingénieur.

La section suivante contient des informations et des astuces importantes à propos de divers aspects d'AVR Libc.

Conseils importants sur AVR Libc

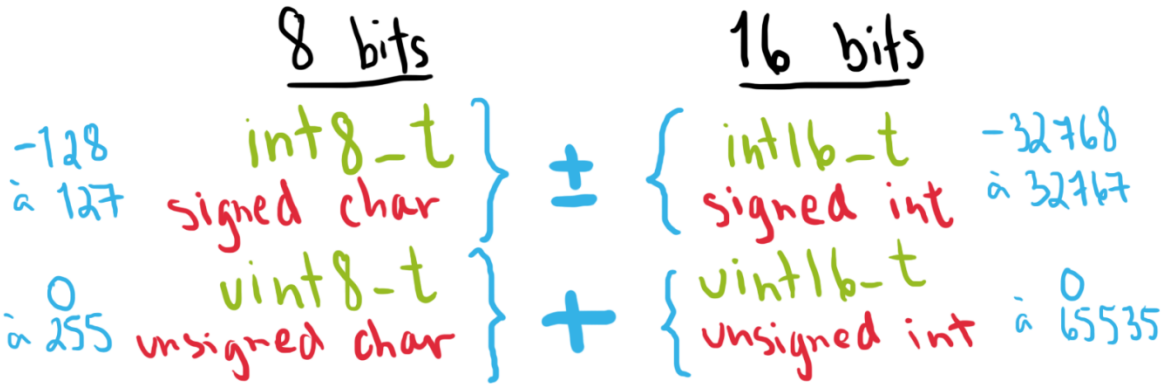
La présente section comprend plusieurs sous-sections décrivant différentes astuces et recommandations quant à l'utilisation d'AVR Libc. En suivant correctement ces directives pratiques, vous devriez vous sauver une bonne pelletée de questions aux chargés et aux répétiteurs, mais surtout vous éviter de recommencer certaines parties des travaux pratiques et du projet final.

Utilisez les bons types

Nous savons qu'en langage C, les types numériques de base comportent entre autres `int`, `char` et `long`. La largeur (le nombre de bits) de ces types dépend de l'architecture visée par la compilation. Sur une architecture IA-32 (parfois nommée « x86 »), un entier de type `int` est réputé mesurer 32 bits. avr-gcc interprète toutefois ce type comme étant sur 16 bits. Afin d'éviter toute confusion, il est préférable d'utiliser les types définis par AVR Libc, de vulgaires `typedef` qui améliorent grandement la lecture du code source puisqu'ils comportent le nombre de bits dans leur nom de symbole :

- `int8_t`
- `uint8_t`
- `int16_t`
- `uint16_t`

Le motif s'étend jusqu'à `uint64_t`, mais, pour des raisons techniques impliquant la taille des registres généraux du microprocesseur de l'ATmega324PA, je vous déconseille de dépasser `uint16_t` dans vos applications. Ces types sont décrits à l'URL <http://www.nongnu.org/avr-libc/user-manual/group__avr__stdint.html>.



Le mot d'ordre est donc : utilisez les types en vert et laissez tomber les types en rouge. N'hésitez pas à *typecaster* si nécessaire; vous aurez à le faire quelquefois entre `char` et `uint8_t` lorsque vous aurez à utiliser des fonctions traitant des chaines.

Ces types sont définis dans l'entête `<stdint.h>` qui sera habituellement automatiquement inclus en incluant n'importe quel autre entête d'AVR Libc.

Utilisez le plus de symboles possibles

Un symbole, en programmation, est généralement un mot non réservé tel que le nom d'une fonction, d'une variable ou d'une définition. AVR Libc procure un joyeux cocktail de définitions et de variables globales qui ont comme mandat principal d'améliorer la lecture de votre code. Je pousserai même l'audace jusqu'à formuler que l'essence d'AVR Libc est sa vaste collection de symboles prédéfinis. Ceci permet de faire abstraction de l'adressage et même de l'indice des bits des registres à modifier! J'explore ici certaines définitions importantes dont quelques unes sont souvent négligées par les étudiants.

- Chaque **registre** d'entrée/sortie a son symbole qui correspond exactement au nom du registre dans la documentation officielle de l'ATmega324PA. Par exemple : `MCUCR`, `PORTC`, `UCSR0B` et `TCCR0A` en sont tous.
- Chaque **bit** de chaque registre d'entrée/sortie a son symbole qui n'est, ultimement, qu'un nombre représentant l'indice du bit dans le registre. Par exemple, le symbole `COM0B1` du registre TCCR0A vaut 5 puisqu'il représente le bit 5 du registre. Ces symboles en particulier sont souvent mystérieusement tenus à l'écart par les étudiants, alors qu'ils sont en réalité très pratiques pour la lisibilité du code source; lire « `TCCR1A |= 2` » est beaucoup moins signifiant que « `TCCR1A |= (1 << WGM11)` ».
- Le dernier point s'applique également aux **ports**. Effectivement, chaque bit de chaque port est défini. Pour le port A, par exemple, on parle des symboles `PA0` à `PA7`, qui valent respectivement 0 à 7. L'avantage est toujours la lisibilité du code source.

Accompagnant les symboles représentant les indices de bits, la macro `_BV` à un paramètre est définie ainsi :

```
#define _BV(bit) (1 << (bit))
```

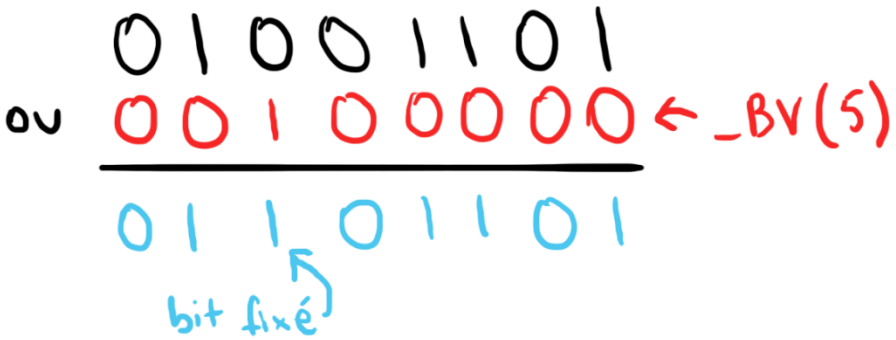
Effectivement que l'opération `(1 << (bit))` était souvent pour obtenir la valeur d'un bit (BV signifie d'ailleurs « *Bit Value* »). Son usage a tendance à décroître… Deux opérations qui paveront littéralement votre code source sont donc :

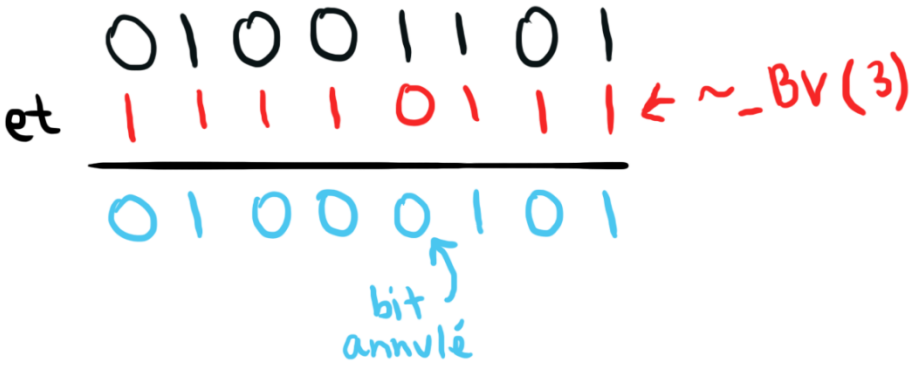
```
UNREGISTRE |= _BV(TELBIT);
```

afin de fixer le bit TELBIT du registre UNREGISTRE à « vrai », ainsi que

```
UNREGISTRE &= ~_BV(TELBIT);
```

afin de le fixer à « faux ». Le dernier échantillon est souvent appelé un « masque », puisqu'il effectue un ET logique avec un masque correspondant à la valeur inverse du bit désiré. Si vous n'êtes pas convaincu de la signification des différents opérateurs, je vous invite à vous rafraichir ici : <http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B>.





Évidemment, fixer plusieurs bits à la fois ne demande que de les « additionner logiquement » (c'est-à-dire d'appliquer un gros OU logique) entre eux :

```
UNREGISTRE |= _BV(TELBIT0) | _BV(TELBIT1) | _BV(TELBIT2);
UNREGISTRE &= ~(_BV(TELBIT4) | _BV(TELBIT6));
```

D'autres définitions intéressantes pour la manipulation de bits incluent `bit_is_set`, `bit_is_clear`, `loop_until_bit_is_set` et `loop_until_bit_is_clear`, toutes décrites à l'URL <http://www.nongnu.org/avr-libc/user-manual/group__avr__sfr.html>.

L'entête `<avr/io.h>` inclut toutes les définitions de symboles décrits dans cette sous-section. Cet entête inclut lui-même par la suite `<avr/sfr_defs.h>`, qui contient les définitions pour la manipulation de bits.

Attention aux délais

Les premiers défis des travaux pratiques en INF1900 comprennent souvent l'utilisation abusive d'une suite de délais. Un délai dit « *busy-wait* » consiste en une boucle qui se répète un certain nombre de fois afin d'occuper (*busy*) le processeur dans le but de créer un délai (*wait*). Ceci signifie que l'exécution de votre code sera bloquée pendant un délai de ce type.

AVR Libc fournit principalement quatre fonctions intéressantes pour l'utilisation de délais :

- `_delay_ms`
- `_delay_us`
- `_delay_loop_1`

- `_delay_loop_2`

Je vous conseille fortement de lire et de comprendre la page descriptive de `<util/delay.h>`, l'entête incluant la déclaration de ces fonctions utilitaires, disponible à l'URL <http://www.nongnu.org/avr-libc/user-manual/group__util__delay.html>. Dans tous les cas, je présente ici de **précieux conseils** en ce qui concerne ces délais.

- `_delay_ms` et `_delay_us` permettent d'attendre respectivement un nombre de millisecondes et de microsecondes. Ceci dit, leur paramètre doit **être connu à la compilation**, ce qui signifie qu'il **ne doit pas être variable**. En d'autres termes, mettez toujours en paramètre une valeur littérale et non le symbole d'une variable. Vous pouvez toutefois y mettre un symbole de définition d'une valeur littérale, puisque ceux-ci sont connus à la compilation. Voici deux bons exemples d'utilisation :

  ```
  #define DELAI 14.37
  _delay_ms(4.5);
  _delay_us(DELAI);
  ```

  Voici maintenant un très mauvais exemple d'utilisation :

  ```
  double a = 15.46;
  a += (double) PINB;
  _delay_ms(a);
  ```

  Le secret réside dans le fait que l'ATmega324PA n'est pas doté d'une unité permettant les calculs de nombres à virgule flottante (représentés en langage C par les types `float` et `double`). Lorsque vous écrivez une valeur littérale en tant que paramètre de ces deux fonctions, le compilateur connait le nombre et l'optimisation fait en sorte que le tout soit en fait un processus opérant sur un nombre entier. Lorsque vous écrivez une variable (qu'elle soit `float` ou non), le résultat n'est pas connu (en général, la conséquence sera d'avoir un délai beaucoup plus grand qu'espéré). Le mot d'ordre : n'utilisez jamais de type `float` ou `double`, nulle part, et ne mettez qu'une valeur littérale comme paramètre de `_delay_ms` et `_delay_us`.

  Si vous voulez réellement une fonction qui exécute un délai en millisecondes variable, codez-la, en implémentant une boucle de 0 à votre **paramètre entier** de millisecondes qui exécute un `_delay_ms` de 1,0. À vous de jouer avec la résolution si la désirez plus grande.

- `_delay_loop_1` prend en paramètre un nombre non signé de 8 bits (`uint8_t`), et « attend » 3 cycles du microprocesseur par incrément. Sachant que la fréquence de l'horloge du microprocesseur est, on se souvient, 8 MHz, il est assez facile en utilisant un ou deux produits croisés de trouver à combien de microsecondes correspond tel nombre en paramètre. L'avantage de l'utilisation de ce délai est qu'il prend un **paramètre variable**. De même, `_delay_loop_2` prend en paramètre un nombre non signé de 16 bits (`uint16_t`), et attend 4 cycles du microprocesseur par incrément. Attention particulière au fait que 0 en paramètre a pour effet spécial d'imposer la valeur maximale plus 1, donc 2^8 pour `_delay_loop_1` et 2^{16} pour `_delay_loop_2`. Si vous voulez réellement un délai nul (de 0)… n'en mettez simplement pas!

- `_delay_ms` et `_delay_us` **doivent connaitre la fréquence du microprocesseur** de l'ATmega324PA (8 MHz). Ceci se fait en définissant `F_CPU` comme étant `8000000UL` :

  ```
  #define F_CPU 8000000UL
  ```

  Assurez-vous de toujours avoir cette définition quelque part (avant l'inclusion de `<util/delay.h>`) lorsque vous utilisez l'un des deux délais. Le fait est que l'optimisation du compilateur « transforme » les instances de `_delay_ms` et `_delay_us` respectivement en `_delay_loop_2` et `_delay_loop_1`. On devine donc maintenant les valeurs maximums de `_delay_ms` et `_delay_us` : il s'agit respectivement du nombre de millisecondes et du nombre de microsecondes pour obtenir $(2^{16} \times 4)$ et $(2^8 \times 3)$, ou 262 144 et 768 cycles d'horloge. En divisant ces deux derniers nombres par la fréquence, 8 000 000 cycles/seconde, nous obtenons respectivement 0,032768 s et 0,000096 s. Ainsi, notre **valeur maximale** pour `_delay_ms` est de 32,768 et de 96,0 pour `_delay_us`.

- La documentation des délais spécifie toutefois que si l'on **dépasse la valeur maximum** du paramètre de `_delay_ms`, la fonction procure une résolution plus petite (donc le délai est moins précis), c'est-à-dire une résolution de 0,1 ms jusqu'à un maximum de 6,5535 secondes, indépendamment de la fréquence d'horloge du microprocesseur. Pour `_delay_us`, dépasser la valeur maximum du paramètre fera en sorte qu'un appel à `_delay_ms` sera plutôt effectué, n'entrainant effectivement aucune conséquence au niveau logiciel, sans avertissement.

Ces derniers points façonnent des considérations très importantes lors de l'écriture d'un logiciel faisant usage des fonctions déclarées dans le fichier `<util/delay.h>`.

Prenez garde à la volatilité des variables globales

Pour des raisons techniques qui dépassent le cadre de ce document, il peut arriver qu'une variable globale modifiée à la fois par la partie standard et la partie interruption (lorsque vous y arriverez) d'un logiciel garde une seule valeur ou présente un comportement surprenant. En gros, l'optimisation du compilateur fait en sorte que les deux modifications dans des sections séparées ne se « voient » pas, désynchronisant ainsi complètement le tout.

L'astuce est de déclarer la variable globale avec le mot-clef standard C `volatile`. Ce mot-clef a pour effet d'indiquer au compilateur qu'il doit considérer que la variable peut être modifiée n'importe quand, n'importe où, et que sa valeur doit toujours être revérifiée plutôt qu'assumée. Voici un exemple :

```
volatile uint16_t var;
```

Utilisez les fonctions de traitement de chaines

Vous aurez parfois à faire différents traitements sur des chaines de texte, qui sont en fait des tableaux de type `char` (donc 8 bits). Plutôt que de réécrire l'équivalent de la

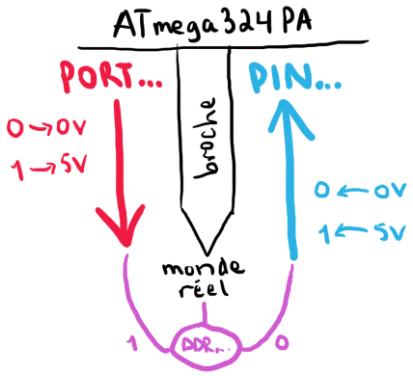
librairie standard C au grand complet, pourquoi ne pas utiliser les fonctions d'AVR Libc déclarées dans le fichier standard `<string.h>`?

Effectivement, vous retrouverez là les bonnes vieilles fonctions `strcat`, `strcmp`, `strlen`, `strcpy` et plus encore! Je vous invite à lire davantage sur les chaines de caractères en langage C et sur leurs particularités sur Wikipedia : <http://en.wikipedia.org/wiki/C_string>, afin de prendre conscience, entre autres, qu'elles se terminent par un caractère nul (un octet de valeur 0).

Faites attention à la direction des ports

Cette question concerne **probablement** davantage la compréhension du microcontrôleur en soi qu'AVR Libc, mais elle est décrite ici quand même. Lorsque vous désirez écrire ou lire d'une broche d'un des quatre ports, assurez-vous des points suivants.

- Chaque broche a une direction : entrée ou sortie. Les variables globales `DDRA`, `DDRB`, `DDRC` et `DDRD` décrivent la direction des broches de chaque port. Les bits 0 à 7 des variables correspondent aux broches 0 à 7 des ports correspondants. Un bit fixé à « vrai » signifie que la broche associée est en sortie, et à « faux » qu'elle est en entrée. Il ne s'agit donc pas d'un port qui est en entrée ou en sortie, mais bien de **chacune des 32 broches individuellement**.
- Pour « écrire » (imposer une tension par rapport à la masse) sur une broche en sortie, on doit utiliser les variables globales `PORTA`, `PORTB`, `PORTC` et `PORTD`. Le même principe que pour les registres de direction s'applique quant à l'indice des bits par rapport à celui des broches.
- Pour « lire » (obtenir une valeur logique à partir des tensions présentes sur les broches), on lit la valeur des variables globales `PINA`, `PINB`, `PINC` et `PIND`. Le même principe que pour les registres de direction s'applique quant à l'indice des bits par rapport à celui des broches. Ces variables correspondent effectivement à des registres d'entrée/sortie différents de l'ATmega324PA. Ainsi, lire de `PORTC` ou écrire dans `PINB` **n'ont aucun effet désiré**. Une erreur commune est de donc de lire `PORTA`, `PORTB`, `PORTC` ou `PORTD` dans le but de lire la valeur logique d'une broche.



Conclusion

Somme toute, la morale du document est qu'on gagne à être **autodidacte** et à approfondir les différentes documentations officielles en INF1900, quitte à trop en lire. On ne compte plus les conséquences positives de la bonne compréhension des

systèmes impliqués : le confort avec les outils, le gain de maturité informatique, les travaux pratiques réalisés plus rapidement et plus linéairement, le projet final armé d'un meilleur design et la lisibilité du code source en sont quelques exemples. Rapidement, vous pourrez vous aussi réaliser l'équivalent de ce document, en plus de connaitre une panoplie d'aspects techniques de l'ATmega324PA et d'AVR Libc.

Ce document subira certainement plusieurs modifications; les corrections de fautes et l'ajout éventuel de contenu auront comme effet d'incrémenter sa version. D'ici là, bonne chance, si vous commencez le cours!