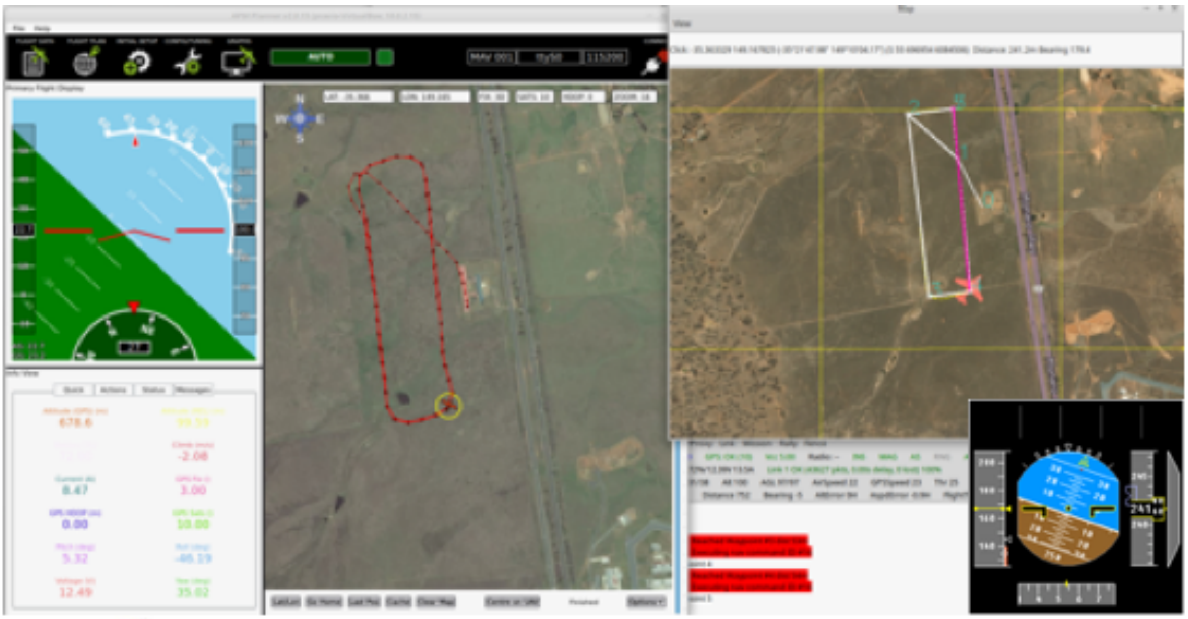
‡

Projet Avionique - A2/S4

‡ ‡ ‡

Gestion & contrôle d'un simulateur de drone



Note des auteurs :

‡–‡

Ce document est en ***version 4.2*** *.... En effet suivant à la lettre l'adage "T'es dans la charrette, pas le temps d'aller sous la couette", cette quatrième version du document que vous avez sous les yeux a vu le jour le dimanche 26/02/2017 à 8h27 GMT+1*[1] *... Aussi, il est fort probable que ce document évolue, i.e. s'enrichisse, dans les jours/semaines à venir ...*

Vous remerciant de votre compréhension ...

‡–‡

Mis à jour le 25 mars 2018

‡–‡

Special Thanks to : SanMarco^{TM}*, Nespresso*^{TM}*, Red Bull*^{TM} *and Guronsan*^{TM}*...*

1. Et est censé être une amélioration de la version précédente ...

Note : les exercices en rouges et entourés de ⋆ sont à faire en séance.

Table des matières

1 Présentation

1.1 Introduction

Ce projet a pour but la prise en main et l'utilisation d'un simulateur de drone (quadricoptère, avion, ...) [2] SITL du projet APM. L'application développée durant ce projet devra permettre d'envoyer des commandes (via un joystick) au simulateur et de récupérer un certain nombre d'informations en provenance de celui-ci afin de les afficher (en utilisant un horizon artificiel (glasscockpit) existant). Deux architectures logicielles seront réalisées :

— une, assez classique et de bas niveau que nous vous aiderons à mettre en œuvre et qui vous permettra de découvrir et d'utiliser différentes notions ;
— une, de plus haut niveau, utilisant ROS, une plateforme (framework) de développement d'application pour Robot pouvant être considéré comme une sur-couche d'un système d'exploitation (Operating System). Après une prise en main de ROS vous devrez réaliser cette architecture de manière plus autonome.

1.1.1 Architecture bas niveau

L'ensemble des éléments manipulés (simulateur, glasscockpit, joystick) étant implanté en langage C/C++, un autre objectif de ce projet est donc de vous donner les bases et de vous familiariser à la programmation dans ces langages, parfois subtiles. La majeure partie du code sera effectué en C. Pour autant ce document comporte un cours d'introduction au langage C++ que les plus téméraires d'entre vous pourront mettre en œuvre si le temps le permet...

Les différentes parties de ce projet communiqueront entre elles via le réseau (communication TCP/UDP/IP), en utilisant le protocole MavLink (Micro Air Vehicle Communication Protocol), très utilisé dans le monde des systèmes embarqués.

La figure 1 synthétise ce que l'on souhaite vous faire réaliser dans ce projet. Votre travail consistera donc en la réalisation de ce qui se trouve dans le nuage de gauche. Nous décrirons de manière détaillée dans une autre partie, l'architecture et le fonctionnement de ce projet.

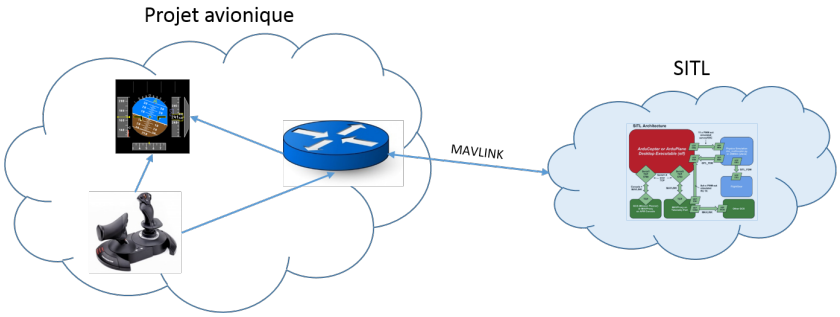


FIGURE 1 – L'architecture bas niveau de notre projet.

1.1.2 Architecture haut niveau

Comme dit plus avant, cette architecture s'appuiera sur ROS.
Sans faire ici une présentation détaillée de ROS [3], il a été initialement développé en 2007 sous le nom "switchyard" par le Stanford Artificial Intelligence Laboratory dans le cadre du projet Stanford AI Robot (STAIR).
Cet environnement peut être vu comme un méta-système d'exploitation pouvant fonctionner de manière distribuée (i.e. sur plusieurs périphériques). Il fournit différentes fonctionnalités de haut niveau, avec entre autres :

— abstraction du matériel,
— contrôle des périphériques de bas niveau,
— mise enœuvre de fonctionnalités couramment utilisées,
— et transmission de messages entre les processus

2. Nous n'utiliserons dans ce contexte que la version avion
3. Puisque cela fera l'objet d'un chapitre plus détaillé dans la suite de ce document

En résumé, ROS regroupe plusieurs fonctionnalités facilitant le développement d'applications modulaire pour la robotique, avec par exemple une architecture de communication inter-processus et inter-machine, un système d'enregistrement et de re-jeu, un système de test et un simulateur (Gazebo).

L'architecture de communication est basée sur la notion de *nodes* et chaque *node* peut communiquer avec d'autres via la notion de *topics*. La connexion entre les *nodes* est gérée par un *master* jounat le rôle de chef d'orchestre entre le *node* souhaitant publier une donnée (*publisher*) et le *node* souahitant recevoir une donnée (*suscriber*). Les messages envoyés sur les *topics* sont pour la plupart standardisés rendant le système très flexible et facilitant l'ajout ou le retrait de *node(s)*.

Vous l'aurez compris, il s'agirat donc dans une deuxième partie du projet de modifier la première architecture en se basant sur ROS pour créer des *nodes* permettant de rendre les fonctionnalités attendues.

1.1.3 En bref ...

Pour terminer cette brève présentation, ce projet a été conçu afin de vous faire non-seulement, découvrir une partie des différentes notions abordées dans l'option Informatique & Avionique, mais aussi de vous faire découvrir/utiliser un certain nombre d'outils/concepts qui vous seront très certainement utiles dans l'avenir, même si, par le plus grand des malheurs, vous ne choisissez pas de nous rejoindre en troisième année ! [4]

1.2 Serveur Linux

Le simulateur SITL (software in the loop) que nous utilisons ici et que nous détaillerons dans la section suivante, permet de réaliser le contrôle/commande complet de drones (volant ou roulant) exécutant un code embarqué basé sur le projet APM (MultiPlateform Autopilot). Ce simulateur, extrêmement complet n'est disponible que pour les systèmes d'exploitation de type Unix/Linux. Vous réaliserez donc ce projet au sein d'un environnement Linux. Pour ce faire nous mettons à votre disposition plusieurs serveurs, précisément un par binôme [5]. Ces serveurs (distribution Linux Mint) sont configurés de manière identique et possèdent les éléments nécessaires (simulateur, glasscockpit, et environnement de développement).

1.2.1 Client/Serveur X

X Window System [6], ou X11 ou encore X est un environnement graphique fenêtré permettant l'affichage et la gestion d'une IHM (écran, clavier, souris, ...). C'est le standard graphique sur tous les systèmes de type UNIX, comme Linux, BSD, mis à part, Mac Os X, pour lequel X n'est pas obligatoire [7].

Le standard X Window System fonctionne selon le modèle client/serveur. Précisément :

- *le serveur X* s'exécute sur une machine possédant un système d'IHM (écran, clavier, souris) et ne nécessitant pas forcément de grosses ressources. Le serveur est à l'écoute, via une communication réseau sur un port logiciel [8], de requêtes d'affichage en provenance d'un client. L'affichage est réalisée par image matricielle (i.e. bitmap). De même il informe le client, toujours via une communication réseau, des différentes interactions de l'utilisateur [9].
- *le client X* n'est ni plus ni moins que le logiciel graphique qui s'exécute sur une machine robuste et puissante, généralement appelée serveur [10], et qui, une fois connecté au *serveur X* lui envoie ses demandes d'affichage en utilisant le protocole X.

Comme vous le voyez, ces deux définitions sont assez troublantes puisqu'allant à l'encontre de ce à qui l'on s'attend : le rôle du *serveur X* est joué par la machine disposant de peu de ressources et inversement pour le *client X* ...

Vous l'aurez compris, cette architecture va donc être utilisée au cours de notre projet. Ainsi, nous exécuterons, sur les machines de la salle de projet, un *serveur X* nous permettant d'afficher les fenêtres des applications

4. Ce qui à notre sens serait une grande erreur ... En effet l'option I&A étant la crème de la crème des options, un Ensmatique qui s'en prive serait comme un Manu sans ses blagues pourries ou un Mikky sans sa clope, voire un élève ENSMA sans le Kaârf ... je m'égard un peu là ...
5. Ou sous groupe de projet une fois ceux-ci constitués.
6. Le première version de ce système a été publiée par le MIT en juin 1984.
7. Puisqu'il utilise Quartz, moteur graphique propriétaire d'Apple.
8. La notion de port logiciel et de communication réseau a dû être vue en première année.
9. Dans ma jeunesse, cet ensemble portait le nom de *terminal X* et était très utilisé puisque ne nécessitant que peu de puissance.
10. Attention, on parle bien ici de serveur et non de *serveur X*

s'exécutant sur le serveur (terminal, simulateur, horizon artificiel, environnement de développement, ...) Linux sur lequel vous serez connecté.
Mais avant cela nous devons nous connecter à l'un de ces serveurs.

1.2.2 SSH - Secure Shell

SSH ou *Secure Shell* est un protocole de communication sécurisée, i.e. crypté, utilisant le port logiciel 22. Le cryptage des données échangées au cours de la communication est assurée par un échange obligatoire de clé de chiffrement en début de connexion.
Ce protocole, qui a également donné son nom à l'application client/serveur l'implémentant, peut-être utilisé dans deux types de situation :

- ouvrir un *shell*, i.e. terminal [11], sécurisé sur une machine distante. Si cette méthode est peu utilisée dans le monde Windows, à contrario, elle l'est de manière intensive dans un environnement Unix/Linux.
- créer un *tunnel* entre deux machines afin d'encapsuler d'autres protocoles de communications utilisant des ports de communication différents. On parle alors de *SSH Tunneling* et/ou de *Port Forwarding*. Par exemple, ceci est notamment utilisé pour sécuriser le protocole de récupération de mail *POP3* qui lui ne l'est pas. Cette méthode peut également s'avérer particulièrement utile dès lors que les deux machines communicantes sont séparées par un firewall n'autorisant pas les ports souhaités.
  Cette deuxième utilisation n'est en fait qu'une extension de la première et se réalise, comme nous le verrons dans la suite, de manière assez simple.

1.2.3 X11 forwarding via SSH

La majorité des logiciels utiles au projet s'exécutant sur le serveur Linux fourni, nous avons donc besoin de réaliser un déport d'affichage (i.e. de l'interface des logiciels utilisés) sur les machines de la salle de projet. Nous utiliserons donc le protocole *X11* et, pour simplifier la communication, nous le ferons via un tunnel *SSH*.

Nous avons donc besoin d'exécuter deux logiciels sur les machines de la salle de projet pour nous connecter à un serveur :

- un *serveur X*
- un client *SSH*

Vous trouverez, sur le moodle du projet, une archive contenant le logiciel MobaXterm. Ce logiciel, dont la version qui vous ai fourni ne nécessite pas d'installation, propose à la fois le *serveur X* dont nous avons besoin, mais aussi le client *SSH* nécessaire à l'établissement de la connexion.

Le fonctionnement du simulateur utilisant de nombreux ports de communication, il est extrêmement délicat d'en exécuter plusieurs instances en parallèle. De plus, ceci nécessiterait des ressources importantes sur le serveur. Aussi nous avons opté pour le mode de fonctionnement suivant. Chacun des binômes d'un groupe de projet se verra remettre, par le Gentil Encadrant, un numéro allant de 1 à 6. A ce numéro correspond une adresse IP (adresse réseau) d'un serveur particulier. Vous trouverez ci-dessous l'association numéro de binôme/Adresse IP. Lors de chaque séance, vous utiliserez le même serveur.

- binôme numéro 1 ==> 193.55.163.212
- binôme numéro 2 ==> 193.55.163.224
- binôme numéro 3 ==> 193.55.163.225
- binôme numéro 4 ==> 193.55.163.227
- binôme numéro 5 ==> 193.55.163.228
- binôme numéro 6 ==> 193.55.163.230

Sur chaque serveur, vous utiliserez le même compte de connexion :

- Login : projavio
- MdP : projavio

L'exercice 1 ci-dessous, va vous permettre de réaliser votre connexion au serveur et de mettre en place le "*montage*" de votre répertoire de travail.

11. Ou encore console.

⋆ Exercice 1 ⋆ : Connexion au serveur Linux

1. *Récupérez l'archive* MobaXterm_v8.6.zip *sur le moodle, dans la section "connexion au serveur Linux", et décompressez là dans un dossier de votre espace de stockage.*
2. *Connexion en mode "plein écran" :*
   (a) *Exécutez le programme* MobaXterm_Personal_8.6.exe
   (b) *Suivre le petit tutoriel présent sur le moodle et correspondant à ce type de connexion*[a].
   (c) *Explorez cet environnement, puis, déconnectez-vous.*
3. *Connexion en mode "fenêtré" :*
   (a) *En suivant le deuxième tutoriel correspondant à ce type de connexion, connectez-vous à votre serveur en mode fenêtré.*
   (b) *lancer un terminal en utilisant la commande* myterm &.
4. *Montage/Démontage de votre répertoire de travail : ces opérations seront à effectuer au début de chaque connexion pour le montage, et avant toute déconnexion pour le démontage.*
   (a) *A la racine de votre compte, accessible via la commande* cd ~, *lister son contenu, à l'aide de la commande* ls -al.
   (b) *Vérifier la présence du répertoire* media. *Si il n'existe pas, créer le par la commande* mkdir media.
   (c) *En utilisant la commande* sudo mount -t cifs //DataEtu/YYYYY media -o username=YYYYY,rw,uid=1001,gid=1001, *où* YYYYY *correspond à votre login sur le domaine w, monter votre répertoire personnel.*
   (d) *Vérifier le contenu de votre répertoire* media (ls ~/media).
   (e) *Si le répertoire* media *contenait bien vos fichiers, vous pouvez maintenant le démonter. Pour ceci, en-dehors de celui-ci, exécuter la commande* sudo umount /home/avio2016/media.
5. *Deconnectez vous par la commande* exit.

a. Pour ceux ne possédant pas les yeux ayant la capacité de déchiffrer du 4K réduit en 320X180, vous pouvez ouvrir la vidéo dans un nouvel onglet de votre navigateur

1.3 Simulateur et Station sol

1.3.1 Simulateur SITL

Nous allons maintenant nous "occuper" du simulateur SITL, déjà installé sur votre système. L'architecture, pouvant vous paraître complexe, de ce simulateur est représentée sur la figure 2.

Comme vous pouvez le constater, et comme cela vous sera détaillé en séance, ce simulateur est composé de trois éléments principaux communiquant les uns avec les autres mais aussi avec l'extérieur par communication réseau TCP/IP :

— le simulateur de code embarqué,
— le simulateur physique,
— la représentation graphique (sur carte) du comportement du drone.

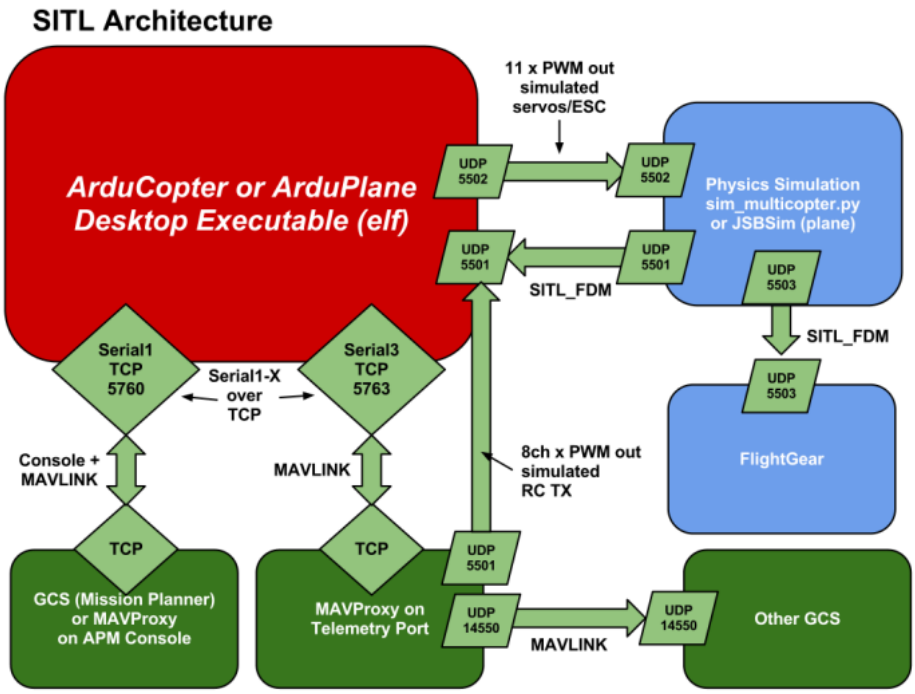


FIGURE 2 – L'architecture du simulateur SITL.

⋆ Exercice 2 ⋆ : Mise en route et test du simulateur SITL

1. *Après avoir ouvert une connexion en mode fenêtré, ouvrir un terminal et placez-vous*[a] *dans le répertoire* Documents/ArduPilot/ardupilot/ArduPlane/.
2. *pour lancer le simulateur et l'ensemble de ses composants, saisir la ligne de commande suivante :* sim_vehicle.sh --console --map --aircraft test[b]
   Deux fenêtres, en plus de la console, doivent s'afficher : la console SITL et la carte.
3. *Dans la console texte, après avoir appuyé sur la touche* ENTREE *vous devez voir apparaître le prompt* MANUAL>. *Taper la commande* help *afin de faire afficher la liste des différentes commandes permettant de contrôler le drone.*
4. *Faire déplacer le drone vers différents points et analyser les différentes informations sur la console SITL.*
5. *Déconnectez vous.*

a. Souvenez vous de la touche TAB ...
b. Dans leur grande générosité, les auteurs vous fournissent les alias *cdsimu* et *startsimu*

1.3.2 Station sol APMPlanner2

Le projet APM propose une station sol (APMPlanner2 cf. figure 3) permettant le contrôle à distance d'un drone physique. Nous allons ici installer cette station sol et l'utiliser afin de contrôler le drone simulé.

FIGURE 3 – La station sol APMPlanner2.

⋆ Exercice 3 ⋆ : Station sol APMPlanner2

1. *Ouvrez une connexion en mode plein écran, et lancer un terminal.*
2. *Dans ce terminal, taper la commande* apmplanner2. *La station sol doit apparaître à l'écran* [a].
3. *Cliquer sur le bouton* Connect *en haut à droite de la station sol et analyser les informations de la station sol.*
4. *Nous allons maintenant charger une mission autonome sur notre drone et le laisser "voyager" en mode autonome.*
   Aller dans la console texte dans laquelle vous avez lancé le simulateur (avec le prompt MANUAL> *et taper la ligne de commande* [b] *:*
   wp load ../Tools/autotest/ArduPlane-Missions/CMAC-toff-loop.txt.
   Taper ensuite la commande AUTO pour faire passer le drone en mode autonome. Analyser.
5. *A vous de jouer ... mais trop quand même ; manipuler drone, console SITL et station sol afin de comprendre tout cet environnement.*

a. Attention, parfois, 2 stations sol peuvent être lancées ...Veillez à en fermer l'une des deux après avoir minimisé les deux fenêtres principales et avoir fermé les petites fenêtres popup

b. Vous pouvez aussi utiliser le Copier/Coller ...

1.4 Emacs & Toolchain

Il est maintenant grand temps de passer aux choses sérieuses ! Vous l'aurez compris, l'ensemble du code à réaliser dans ce projet s'implantera en langage C [12]. Pour ce faire nous utiliserons deux types d'outils :

- une éditeur de code : dans notre cas il s'agira ici de l'éditeur *Emacs*,
- une toolchain : ici la suite d'outils *gcc* que nous emploierons :
  - via la ligne de commande dans un premier temps,
  - cia un Makefile, dès lors que le projet deviendra un peu plus volumineux.

12. Et éventuellement C++ sil le temps le permet ...

1.4.1 La Toolchain GCC

L'exécutable gcc est un programme permettant d'appeler le préprocesseur, le compilateur, l'assembleur et le linker pour les langages C et C++. Les opérations réalisées sont les suivantes :

- appel du préprocesseur C (nommé cpp) ;
- appel du compilateur C (nommé cc1), ou C++ (cc1plus) selon le type de fichier à compiler (les fichiers .c sont compilés en C, les fichiers .C, .cc et .cpp sont compilés en C++). Les compilateurs génèrent un fichier assembleur ;
- appel de l'assembleur (as) pour générer le fichier objet, si l'option -S n'est pas spécifiée ;
- appel de l'éditeur de liens (ld) pour générer l'exécutable ou la bibliothèque, si l'option -c n'est pas spécifiée. Dans ce cas, la librairie C standard est incluse par défaut dans la ligne de commande de l'édition de liens.

Le programme gcc accepte différentes options. Voici quelques unes des options les plus fréquentes :

Option	Description
–help	Affiche l'aide de gcc.
–version	Donne la version de gcc.
-E	Appelle le préprocesseur. N'effectue pas la compilation.
-S	Appelle le préprocesseur et effectue la compilation.
-c	Appelle le préprocesseur, le compilateur et l'assembleur, mais pas l'éditeur de lien. Seuls les fichiers objets (.o) sont générés.
-o nom	Spécifie le nom du fichier objet généré.
-w	Supprime tous les warnings.
-W	Active les warnings supplémentaires.
-Wall	Active tous les warnings possibles.

1.4.2 L'éditeur Emacs

Certes, il ne s'agit certainement pas de l'IDE le plus connu lorsque l'on travaille dans un environnement Windows. À contrario, il s'agit de l'un des éditeurs les plus connus et utilisé dans le monde UNIX[13]. *Emacs*, crée par Richard Stallman, est un éditeur composé d'un ensemble extensible de fonctionnalités codée en utilisant le langage d'extension Emacs Lisp. Cette rencontre risque peut-être d'être délicate au début, mais avec un peu d'apprentissage il deviendra peut-être votre éditeur favori ...

Comme vous pourrez le constater dans un instant, cet éditeur vous permettra également de gagner en souplesse et dextérité au niveau des phalanges ...
En effet, Emacs utilise deux types de commandes : les commandes simples, accessibles via les raccourci débutant par la touche Ctrl (que nous noterons C- dans la suite), et les commandes étendues, accessible via les raccourcis débutant par la touche méta ou touche Esc (que nous noterons M- dans la suite).
Voici ci-dessous un très (très très) petit extrait des raccourcis utiles lors de l'utilisation d'Emacs.

Ouverture :

- lancer Emacs : *emacs*
- ouvrir un fichier *f* : *emacs f*

Quitter Emacs :

Commande	Commande étendue	Description
C-z	M-x suspend-emacs	Suspendre (ou iconfier quand on est en mode graphique) emacs
C-x C-c	M-x quit-window	Quitter emacs

Aide d'Emacs :

Commande	Commande étendue	Action
C-h	M-x help	Aide d'emacs (M- ? pour la config conscrits 2002)
C-h k	M-x describe-key	Brève description d'une commande
C-h i	M-x info	Lance les fichiers d'aide info.
C-h m	M-x describe-mode	Description d'un mode majeur ou mineur
C-h t	M-x help-with-tutorial	Lance le tutorial d'emacs

13. Son principal concurrent dans cet univers est l'éditeur *vim*

Manipulation des fichiers et buffers : Emacs ouvre un fichier dans un *buffer* et peut ouvrir plusieurs *buffer* en parallèle.

Commande	Commande étendue	Action
C-x C-f	M-x find-file	Ouvrir un (nouveau) fichier
C-x C-s	M-x save-buffer	Sauvegarder le buffer courant
C-x s	M-x save-some-buffers	Sauvegarder tous les buffers en cours d'édition
C-x C-b	M-x list-buffers	Avoir la liste de tous les buffers.
C-x b	M-x switch-to-buffer	Changer de buffer
C-x C-q	M-x vc-toggle-read-only	Passer le buffer en lecture seule , ou lecture-écriture (selon l'état de départ)
C-x o	M-x other-window	Passer à une autre fenêtre
C-x 1	M-x delete-other-windows	Faire disparaître toutes les fenêtres sauf la fenêtre courante
C-x 2	M-x split-window-horizontally	Partage la fenêtre courante en 2, horizontalement
C-x 3	M-x split-window-vertically	Partage la fenêtre courante en 2, verticalement

Effacer du texte :

Commande	Commande étendue	Action
C-d	M-x delete-char	Efface le caractère sur lequel est le curseur.
M-d	M-x kill-word	Efface le mot à partir du curseur.
M-backspace	M-x backward-kill-word	Efface le mot précédent.
C-k	M-x kill-line	Efface la ligne à partir du curseur

Sélectionner du texte :

Commande	Commande étendue	Action
C-espace	M-x set-mark-command	Poser une marque
M-h	M-x mark-paragraph	Sélectionner tout le paragraphe
C-x h	M-x mark-whole-buffer	Sélectionner le buffer entier

Couper/Copier/Coller :

Commande	Commande étendue	Action
C-w	M-x kill-region	Couper la sélection
M-w	M-x copy-region-as-kill	Copier la sélection
C-y	M-x yank	coller

Chercher/Remplacer :

Commande	Commande étendue	Action
C-s	M-x isearch forward	Recherche simple vers la fin du fichier
C-r	M-x isearch backward	Recherche simple vers le début du fichier
M-%	M-x query-replace	Remplacer

Interaction avec le shell :

Commande	Commande étendue	Action
M-!	M-x shell-command	Exécute une commande shell
	M-x shell	Lance un shell dans terminal rudimentaire (sans séquences d'échappement)
	M-x term	Lance un terminal plus élaboré
C-u M-!		Insère le résultat d'une commande dans le buffer courant

Divers :

Commande	Commande étendue	Action
C-_ ou C-x u	M-x undo	Annule la dernière action
C-g	M-x keyboard-quit	Annule une commande en cours de frappe ou d'exécution
M-q	M-x command fill-paragraph	Reformate le paragraphe
M-g <n>	M-x goto-line	Va à la ligne <n>
	M-x doctor	Le psychanalyste d'emacs. Emacs est votre ami.
	M-x handwrite	Transforme votre fichier texte en PostScript prêt à imprimer, avec une écriture manuscrite

Voilà ! vous êtes prêts ? ? ? ? Alors allons-y ...

⋆ Exercice 4 ⋆ : Mon premier projet C dans Emacs

1. *Ouvrez une connexion en mode fenêtré, et lancer un terminal.*
2. *Montez votre dossier personnel dans le dossier* media.
3. *Déplacez vous dans votre dossier personnel et créez un dossier* ExoC.
4. *Dans ce dossier, créez et ouvrez, à l'aide la commande* emacs hello.c &, *le fichier* hello.c.
5. *Une fois apparue la fenêtre de l'éditeur Emacs, taper le code suivant dans ce nouveau fichier :*

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Coucou les A2 du projet avionique !!!\n");
  return 0;
}
```

6. *Construction et exécution :*
   (a) *Déplacez vous dans le terminal et compiler votre fichier* hello.c *de telle sorte à obtenir un exécutable* hello.out.
   (b) *Exécutez le fichier* hello.out.

2 Le Langage C

Cette partie vous présente les notions de langage C nécessaires pour la bonne réalisation de votre projet. Différents petits exercices sont proposés, vous permettant de mettre en œuvre ces notions.
Les auteurs font ici l'hypothèse que vous maitrisez déjà un certain nombre de notions en langage C (structures d'exécution, types primitifs et composés, sous-programmes et pointeurs) ...
Mais, dans leur infinie bonté, les auteurs consentent tout de même à faire ci-dessous quelques rappels de langage C ...

2.1 Structure d'un programme C

Structure d'un programme C :
- Directives au préprocesseur
- Construction de types
- Déclaration de variable(s)
- Déclaration de fonction(s)
- Définitions de variable(s)
- Définitions de fonction(s)

Un exemple simple :

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        printf("Coucou les gens !!!\n")
        return 0;
}
```

En somme :
- Un programme est une collection de fonctions et d'éventuelles variables globale
- Question : ça commence à quel endroit ? ? ?
  - Réponse : la fonction *main*

La fonction *main* :
- Point d'entrée du programme
- Doit toujours être présente
  - sinon erreur lors de l'édition des liens
- L'exécution débute à la première ligne de cette fonction

- Attention, c'est une fonction comme une autre ...
  - Les variables déclarées à l'intérieur sont locales

2.2 Syntaxe du langage C

2.2.1 Les commentaires

- Deux types
  - /* ... */ : bloc de commentaires
  - // : ligne de commentaires (vient du C++)

2.2.2 Mots clés :

Voici la liste de ceux-ci :

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

2.2.3 Les identificateurs :

- sensible à la casse
- suite de lettre et de chiffre
- premier caractère est obligatoirement une lettre
- _ est considéré comme une lettre
- par convention, ne pas l'utiliser ; réserver à la création de bibliothèque

2.2.4 Opérateurs simples

Voici la liste de ceux-ci :

```
( ) [ ] . ! ~ < > ? :
= , + - * / % | & ^
```

2.2.5 Opérateurs composés

Voici la liste de ces opérateurs :

```
-> ++ -- <= >= == != && || << >>
+= -= *= /= %= <<= >>= |= &= ^=
```

2.2.6 Constantes littérales : nombres entiers

- définis de manière classique et sans signe
- base : décimale par défaut
  - octale (préfixé par 0)
  - hexadécimale (préfixé par 0X ou 0x)
- type :
  - plus petit type dans lequel la constante est représentable
  - en suffixant la constante il est possible de forcer le compilateur :
    - U, u : force le type unsigned
    - L, l : force le type long
    - possibilité de combiner les deux

2.2.7 Constantes littérales : nombres flottants

- constitués par :
  - suite de chiffres décimaux
  - un point
  - suite de chiffres décimaux
  - E ou e
  - signe + ou -
  - suite de chiffres décimaux
- règle d'écriture :
  - on peut ne pas renseigner :
    - la partie entière ou la partie décimale
    - le point ou l'exposant
  - exemple :
    - .12E7 ; 53.e22 ; 48532. ; 10e6
- type :
  - double par défaut
  - possibilité de suffixer pour forcer le compilateur :
    - F ou f : float
    - L ou l : long double

2.2.8 Constantes littérales : caractères et chaînes de caractères

- caractère : entre cote
  - 'A'
- chaine de caractères : entre guillemet
  - "coucou"
- n'importe quel caractère
- séquences d'échappement

```
\n      Saut de ligne
\t      Tabulation
\b      Epace arrière
\r      retour chariot
\f      Saut de page
\a      Sonnerie
\\      \
\'      '
\"      "
```

- représentation des caractères
  - entier sur un octet : le type char
  - nombre représentant le caractère de manière interne
    - ASCII
- représentation des chaîne de caractères
  - suite finie de caractères
  - longueur quelconque
  - codage interne :
    - caractères rangés de manière contiguë
    - ajout d'un caractère nul après le dernier caractère utile
  - donc adresse de la mémoire ou a été "stocké" le premier caractère de la chaîne
  - en bref :
    - Pas de type chaîne prédéfini
    - Utilisation d'un tableau de caractères

2.3 Les types

2.3.1 Types simples

Ci-dessous l'ensemble des types simples connus par le compilateur C.

- Types entiers

Type entier :	Modélise :
char	Entier sur 8 bits
short	Entier sur 16 bits
int	Représentation entière la plus efficace
long ou long int	Entier sur 32 bits

— Types réels

Type réel :	Modélise :
float	Réel simple précision
double	Réel double précision
long double	Réel très grande précision

— Type nommés (description par extension)

Type :	Modélise :
enum	Enumération où chaque élément est associé à un entier

— Type "tout"

Type :	Modélise :
void	Tout ou vide

2.3.2 Modificateurs de type :

— Possibilité de modifier un type :
  — unsigned : version non signé du type
    — Exemples :
      `unsigned int entier non signé de la taille du mot machine:` $(0..2^n - 1)$
      `unsigned seul est un raccourci pour unsigned int`
      `unsigned char code caractère non signé (correspond au type character Ada): 0..255`
      `unsigned short (0..65535) unsigned long` $(0..2^{32} - 1)$
  — long : version longue du type
    — Exemples :
      `long int est l'exception (équivalent à long)`
      `long double flottant IEEE 754 sur 80 bits`

2.3.3 Types construits :

ci-dessous l'ensemble des types construits connus par le compilateur C :

— Type tableau

Type :	Modélise :
[ ]	Tableau

— Type Union

Type :	Modélise :
union	Union : Ou d'éléments variables

— Type Structure

Type :	Modélise :
struct	Ensemble d'éléments variables

— Type Pointeur

Type :	Modélise :
*	Pointeur sur la variable : i.e. adresse mémoire

— Type fonction

Type :	Modélise :
()	fonction

Il existe aussi une instruction permettant de nommer un type : typedef

— `typedef unsigned char octet;`

— le type octet correspond désormais à un entier non signé sur 8 bits

2.3.3.1 Enregistrement :

Création et utilisation d'un type enregistrement :

— création :

```
typedef struct {
        int reel;
        int imaginaire;
} Complexe;
Complexe c1,c2;
```

— utilisation

```
c1.reel = 3;
c2 = c1;
```

— Attention c1 == c2 est interdit ...

2.3.3.2 Union : Le type union permet d'utiliser une zone mémoire en la considérant en fonction du contexte comme d'un type ou d'un autre :

— enregistrement dont tous les membres seraient superposés

```
typedef union {
        char c;
        unsigned long i;
} char_int;
/* L?union utilise 4 octets=max(1,4) */
char_int c;
unsigned long i;
c.c = 'a'; /* c.c est un caractère */
i=c.i; /* c.i vaut alors 0x00000061 (valeur Hex de 'a' */
```

2.3.3.3 Énumération : Description d'un ensemble par extension

— `typedef enum {trefle, carreau, c?ur, pique} couleur;`

— Comme en Ada, correspondance entre valeur de type et index (entier)
  — trefle correspond à 0, carreau à 1, etc.

2.3.4 Déclaration de variables :

— Localisation :
  — si en dehors de toutes fonctions :
    — variable globale
  — si au début d'un bloc
    — variable locale ; visible dans le bloc et les sous blocs
  — en tant qu'argument de fonction
    — variable formelle, i.e. globale
— Syntaxe :
  `[Op_porté] type id [=valeur] ;`
— Exemple :

```
#include <stdio.h>
//Déclaration de variables ...
int numerique; //un entier
float x,y,z = .5f; //3 réels initialisés à 0.5
char c = 'd'; //un caractère initialisé à d
char chaine [] = "Coucou les gens" //une chaine de caractères initalisée
int monTab[3]; //un tableau de 3 entiers
int saispas [3][5]; //à votre avis ??
```

2.3.5 Porté et visibilité des variables :

— Variables statiques :
  — `static`
  — permanente ; existe durant toute la vie du programme
  — les variables globales sont toujours statiques
— variables registres
  — `register`

  - à utiliser si accès très fréquent durant l'exécution
  - peuvent ainsi être placées dans des registres du CPU
- Variables constantes et volatiles
  - `const`
  - pas d'évolution de la variable au cours de l'exécution du programme
  - vérification à la compilation

2.3.6 Initialisation :

- pas de restriction sur la valeur utilisée
- toujours initialiser une variable avant sa première utilisation

2.4 Les pointeurs

2.4.1 Déclaration et utilisation :

Les pointeurs sont fondamentaux en C :
- Un pointeur contient une adresse.
- La déclaration en préfixant le nom de la variable par *

```
int * pt_entier; char c,* pt_char;
```

- L'accès à l'adresse d'une variable pointeur se fait par l'opérateur &

```
int i=3;
pt_entier = &i;
```

- L'accès à la valeur pointée contenue dans la variable se fait par l'opérateur *

```
int j= *pt_entier; /* j vaut dorénavant 3 */
*pt_entier = 4; /*la valeur pointée par pt_entier vaut 4, de même que i*/
```

- La maladie du programmeur C :
  - Utiliser les void * sans aucune notion du type de la valeur pointée ...

2.4.2 Allocation dynamique :

Avant de pouvoir stocker une valeur dans une variable "pointeur", il faut allouer de la mémoire :
- un espace suffisant (i.e. de la taille du type de la variable pointée)
  - en utilisant la fonction sizeof(type)
- on utilise la fonction void *malloc(size_t elsize)
- la fonction renvoie un type void * que l'on doit "caster" (i.e. préciser le type) si l'on veut éviter un avertissement

```
int * pt_entier;
pt_entier=(int*)malloc(sizeof(int));
typedef struct {
        int i;
        char j;
} ma_struct;
ma_struct *pt_ms=(ma_struct*)malloc(sizeof(ma_struct));
```

- Une zone allouée avec malloc (ou calloc Cf.suite) est libérée avec free

```
free(pt_entier);
free(pt_ms);
```

  - indispensable sous peine de saturer la mémoire
  - mais dangereux ... il ne faut pas tout effacer ....

2.4.3 Tableaux : le retour ...

un tableau C est défini comme l'adresse de début d'une zone de mémoire stockant de façon contiguë un certain nombre de valeurs du même type (i.e. de même taille)

```
int T[5];
```

- Attention les indices vont forcément de 0 à n-1
- un tableau ne connaît pas sa taille (donc ses indices non plus)

— Quand on déclare un tableau d'une taille donnée :
  — l'espace contigu nécessaire est alloué
  — la valeur du tableau est l'adresse du début de cette zone allouée
  — **un tableau est donc un pointeur**

```
T[i] //fait référence à ième case (à partir de 0)
T[3]=2; i=T[1]; for (i=0;i<5;i++) { T[i]=0; }
&T[0] //est l?adresse du début du tableau (i.e. adresse de la première case)
&T[i] //est l?adresse de la ième case du tableau
int *PT=&T[0]; //PT pointe sur le tableau
T[6]=3; //faux mais licite ... pas d?exception à l'exécution comme en Ada !!!
```

Un tableau étant un pointeur, l'affectation et la comparaison se font sur les **pointeurs** et non sur **les contenus** :

```
int T[3]={0,2,4};
int T2[3]={0,2,4};
T==T2 //renvoie 0 (faux) car T et T2, même contenu mais pas la même valeur d'adresse du
    tableau
int T3[3]={0,3,5};
T3=T; //affecte à T3 l?adresse de T ; T et T3 pointent sur le même tableau
```

Manipulation :

— Comparer deux tableaux de même taille : memcmp
  — Les attributs de tableaux et les tests de débordement n'existent pas en C.
  — attention, l'opérateur sizeof donne la taille du pointeur, pas la taille du tableau

```
memcmp(T,T2,3*sizeof(int))
/* renvoie une valeur non nulle (vrai) si les valeurs contenues
dans T et T2 sont les mêmes*/
```

— Affecter deux tableaux : memcpy
  — la taille du tableau destination doit être suffisamment grande
    — sinon conséquences difficilement prédictibles

```
memcpy(T,T2,3*sizeof(int)) //copie le contenu de T2 dans T.
```

— Allocation dynamique de tableau : malloc ou calloc
  — exemple :

```
int *T;
T=(int *)calloc(5,sizeof(int)); //Alloue un tableau de 5 éléments entiers
/*Attention, dans ce cas ne pas oublier de faire free(T)
Noter la différence calloc/malloc: calloc initialise tous les octets alloués à 0*/
...
free(T);
```

2.5 Entrées/Sorties

2.5.1 Sorties écran :

utilise la fonction printf et les indicateurs de format.

— printf
  — Syntaxe :
  `printf("...", var1, var2 ... )`
  — Indicateurs de format : à utiliser pour "insérer" des valeurs de variables au sein de la chaine

Indicateur	Affichage
%c	Caractère
%d	Décimal
%x	Hexadécimal
%f	Réel
%.2f	Réel avec 2 chiffres après la virgule
%s	Chaine de caractères
...	

  — Nécessite la bibliothèque stdio.h

2.5.2 Entrées clavier :

utilise la fonction *scanf* et les indicateurs de format.

— *scanf*

— Syntaxe :
— `scanf("Indicateur", &var)`
— & représente l'adresse de la variable
— Nécessite la bibliothèque stdio.h
— exemple :

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        int nombre;
        printf("Saisir un nombre entier :");
        scanf("%d",&nombre);
        printf("\nLa valeur saisie est %d",nombre);
}
```

⋆ Exercice 5 ⋆ : pour s'échauffer un peu ...

1. *Toujours dans votre répertoire personnel, créez un nouveau répertoire pour cet exercice. Dans celui-ci créez et ouvrez à l'aide d'Emacs un nouveau fichier portant l'extension .c.*
2. *Écrire un programme demandant un entier et un réel, en effectuant la multiplication et en affichant le résultat.*
   Compiler et construire en ligne de commande.
   Tester.
   Que se passe-t-il si l'on saisit le réel d'abord ? Pourquoi.
3. *Créer et ouvrez un nouveau fichier portant l'extension .c*
   Écrire un programme qui affiche en hexadécimal la valeur d'une variable saisie en décimale et inversement.

2.6 Structures de contrôle

2.6.1 Conditionnelle : Si ... Alors ... Sinon

— Syntaxe :

```
...
if (condition) {
        //bloc d'instructions du then
} else {
        //bloc d'instruction du else
}
```

2.6.2 Opérateurs de comparaison & logiques :

— comparaison :

Opérateurs comparaison	Signification
==	Vrai si égal
!=	Vrai si différent
>	Vrai si supérieur
>=	Vrai si supérieur ou égal
<	Vrai si inférieur
<=	Vrai si inférieur ou égale

— logiques :

Opérateur logique	Signification
!	Négation logique
&&	ET logique
||	OU logique

— Attention : pas de type booléen en C ...
— 0 correspond à FAUX

  - autres valeurs correspondent à VRAI
- Attention : erreur fréquente
  - Confusion entre = et ==
  - Confusion entre opérateurs logiques et opérateurs binaires

2.6.3 Conditionnelle multiple : si valeur de ... choix

- Syntaxe

```
Switch (variable){
  case valeur : instructions; ...
    break;
  case valeur2 : instructions; ...
    break;
  ...
  ...
  default : instructions_par_defaut ;
}
```

2.6.4 Structures itératives

- `For` :
  - Syntaxe :

```
for (initialisation;condition arret;opérations){
  instructions;
  ...;
  ...;
}
```

- `While` :
  - Syntaxe :

```
while (condition) {
  instructions;
  ...;
  ...;
}
```

- `Do ... While` :
  - Syntaxe :

```
do {
  instructions;
  ...;
  ...;
}
```

2.7 Sous-programmes

Il ne doit y avoir qu'une seule fonction main dans un programme C : c'est le programme principal. Il par contre possible de définir des sous-programme qui, en C, sont obligatoirement des fonctions.

2.7.1 Déclaration

La signature d'un sous-programme est :
- type_de_retour nom_fonction (paramètres) ;
- si aucun paramètre, mettre des parenthèses vides
- si aucun type de retour (i.e. une procédure), utiliser le type void pour le type de retour (tout sous-programme est une fonction)
- Corps du sous-programme :

- type_de_retour nom_fonction (paramètres) {
  définitions (avec ou sans initialisation) des variables locales
  corps
  }
- Tous les paramètres sont en mode IN

```
void echange(int a, int b) {
  int c=a;
  a=b;
  b=c;
}
...
echange(x,y);/* les valeurs n?ont pas été échangées */
```

2.7.2 Passage de paramètres

Comme vu ci-dessus, par défaut le mode de passage des paramètres est IN
Pour obtenir un mode de passage en OUT ou IN OUT :

- il faut passer les paramètres par pointeur :

```
void echange(int *a, int *b) {
  int c=*a;
  *a=*b;
  *b=*c;
}
...
echange(&x,&y);/* les valeurs ont été échangées */
```

- Attention, les tableaux sont déjà des pointeurs ...
  - inutile de passer un pointeur sur un paramètre de type tableau (qui serait un pointeur sur un pointeur) sauf si l'on souhaite modifier l'adresse du tableau ...

```
void inverser(int T[], int tailleT) {
  int i,tmp;
  for (i=0;i<tailleT/2;i++) {
        tmp=T[i];
        T[i]=T[tailleT-1-i];
        T[tailleT-1-i]=tmp;
}
...
inverser(T,10); /* le tableau est inversé */
```

  - Le mot clé const peut être utilisé sur un paramètre
    - le compilateur vérifie s'il n'y a pas de modification du paramètre dans la fonction
    - permet de passer un tableau en mode IN

```
void f(const int T[], int tailleT) { ... }
```

    - Certains compilateurs ne donnent qu'un Warning en cas de modification d'un paramètre constant dans le corps du sous-programme
- Si une fonction doit retourner un tableau ou une adresse : attention à l'allocation mémoire

```
int * tab_10() {
  int T[10],i;
  for (i=0;i<10;i++) {
        T[i]=i;
  }
  return T;
}
/* T étant une variable locale, l ?espace mémoire est désalloué
dès la fin de la fonction. */

int main() {
  int i,*T=tab_10();
  for (int i=0;i<10;i++) {
        printf("%d ",T[i]);
  }
  return 0;
}
```

- il faut réaliser l'allocation mémoire avant l'appel de la fonction

```
...

int main(int argc, char* argv[]) {
  int *T=(int*)calloc(10,sizeof(int));
  T=tab_10(T); // fonctionne également avec tab_10(T) ...
  for (int i=0;i<10;i++) {
        printf("%d ",T[i]);
  }
  return 0;
}
```

2.8 Modularité

Faisons appel à votre mémoire infaillible : le TP d'Introduction aux Systèmes Embarqués sur les feux tricolores ... Vous y êtes ? ? Bien, alors vous vous souvenez surement d'avoir utiliser un module C composé de deux fichiers : feux.h et feux.c [14]. Et bien c'est un très bon exemple de ce qu'il faut faire.

2.8.1 Structure d'un module

Un module C est composé de deux fichiers, un peu comme un module Ada (i.e. package). Plus précisément pour un module MonModule :

- un fihier MonModule.h contenant généralement la spécification du module (comme le .ads Ada), c'est-à-dire les déclarations de types, déclarations de fonctions, ...
- un fichier MonModule.c contenant, au minimum, l'implantation des fonctions déclarées dans le fichier .h (comme le .adb en Ada). D'autres fonctions peuvent être ajoutées, mais elles ne seront alors pas visibles depuis l'extérieur du module. En effet, il n'y a pas de restriction définie par le langage sur le contenu de ces deux fichiers.
  Enfin, le fichier MonModule.c doit faire l'inclusion du fichier MonModule.h à l'aide de la directive préprocesseur #include ....

Mais attention, contrairement au langage Ada, tout ceci n'est, en C, que convention.

2.8.2 Espace de nommage & inclusion

Une différence majeure de fonctionnement entre des langages comme Ada, Java, ... et C provient de la gestion des espaces de nommage. Un espace de nommage est un contexte dans lequel les entités (types, variables, fonctions, ...) sont définies. Dans de nombreux langages, un espace de nommage est associé à un module/package, contrairement au langage C où cette notion n'existe pas. Lors d'inclusions multiples (plusieurs #include du même fichier), il risque donc fort d'y avoir des conflits de noms. Par exemple, si deux types de même nom son définis dans deux modules différents et que ces derniers sont utilisés (i.e. inclus) dans un troisième fichier, le compilateur renvoie une erreur du type conflicting types for ... ou ... multiply defined.
Pour éviter ces erreurs voici, ci dessous la structure de header (fichier .c) à respecter :

```
#ifndef MONMODULE_H_
#define MONMODULE_H_

/* Déclaration des types, fonctions, ... */

#endif/*MONMODULE_H*/
```

2.8.3 Compilation & utilisation

Une fois le module (i.e. fichier .h et .c) créé et compilé, il peut-être utilisé dans un autre programme. La compilation d'un module produit un fichier .o qui sera utilisé lors de la construction (Build) du programme. La figure 4 décrit ce processus.

2.9 Directives préprocesseur

Fonctionnement :

- commencent toujours par :
  #
- lues et interprétées par le processeur avant de commencer la phase de compilation

14. Réalisé par notre Gourou à nous : Manu !

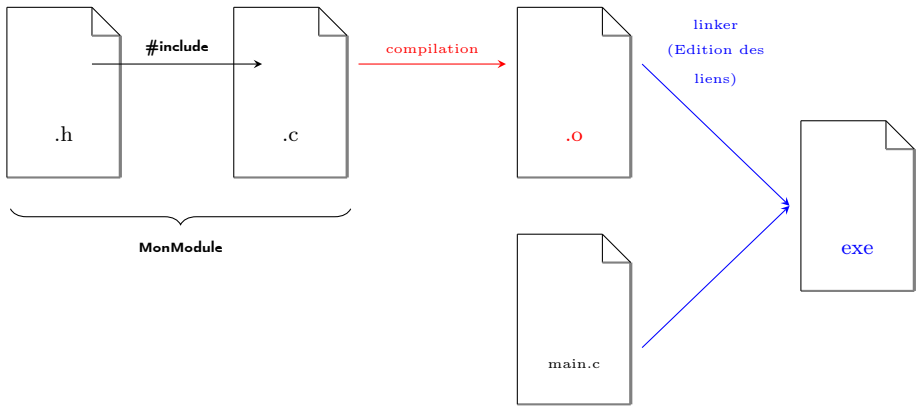


FIGURE 4 – Compilation, construction & utilisation d'un module.

Utilité ?

- Inclusion d'en-tête (fichier .h) (équivalent au with/use Ada)
  - `#include`
  - Simple recopie du fichier .h dans le fichier .c appelant.
    - Attention aux doubles définitions ...
- Définition de constantes :
  - `#define`
  - Association d'une valeur à un mot
- Possibilité de définir une opération au moment de la déclaration des constantes préprocesseurs :
  - opération logique
  - opération binaire
  - ... etc
- Exemple :

```
#define NOMBRE_1 0X12
#define NOMBRE_2 15
#define SOMME_N1_N2 (NOMBRE_1 + NOMBRE_2)

int main(int argc, char *argv[]) {
  printf("Valeur : %d",NOMBRE_1);
  printf("Valeur : %d",SOMME_N1_N2);
}
```

⋆ Exercice 6 ⋆ : Produit scalaire de deux vecteurs

1. *Créez un programme principal permettant de réaliser le produit scalaire de deux vecteurs. Ce programme devra :*
   - *modéliser un vecteur de façon dynamique ; i.e. la taille des vecteurs est demandée à l'utilisateur en début de programme.*
   - *demander, pour chacun des deux vecteurs, chacune des valeurs en fonction de la taille saisie par l'utilisateur.*
   - *réaliser et afficher le produit scalaire de ces deux vecteurs.*

Exercice 7 : Module calcul matriciel

Créer un module calcul matriciel[a]*. On rappel ici qu'un module est composé de deux fichiers : un fichier* monModule.h *contenant les parties déclaratives et un fichier* monModule.c *contenant les réalisations des sous-programmes du module.*

- *il devra contenir la définition d'un type Matrice contenant*
  - *un tableau à deux dimensions,*
  - *deux entiers donnant la dimension de la matrice*
- *il fournira les opérations suivantes :*
  - *Création d'une matrice (allocation mémoire)*
  - *Destruction d'une matrice (désallocation)*
  - *Saisie d'une matrice (dimensions, puis contenu)*
  - *Affichage d'une matrice*
  - *Addition de deux matrices*
  - *Multiplication de deux matrices*
- *Le programme principal permettra de saisir deux matrices, de les additionner et d'afficher la matrice résultante*

a. On nommera ce module Matrice

3 Programmation réseau

Nous nous intéressons ici à la programmation d'application communicante via un réseau.

3.1 Les sockets : principes

La programmation réseau se base sur la notion de *socket*. C'est cette notion, ainsi que celles associées, sue nous allons découvrir dans la suite.

3.1.1 Introduction

La notion de sockets a été introduite dans les distributions de Berkeley (système de type UNIX très connu, dont beaucoup de distributions actuelles utilisent des morceaux de code), ce qui explique pourquoi on parle parfois de *sockets BSD* (Berkeley Software Distribution).

Pour faire simple, il s'agit d'un modèle permettant la communication inter processus (IPC - Inter Process Communication) afin de permettre à divers processus (pouvant être écrits dans différents langages) de communiquer aussi bien sur une même machine qu'au travers d'un réseau en se basant sur le protocole TCP/IP défini dans le modèle OSI.

La communication par socket est souvent comparée aux communications humaines. En effet, on peut distinguer deux types de communication :

- Le mode connecté (qui est comparable à une communication téléphonique), utilisant le protocole TCP. Dans ce mode de communication, une connexion durable est établie entre les deux processus, de telle façon que l'adresse de destination n'est pas nécessaire à chaque envoi de données.
- Le mode non connecté (analogue à une communication par courrier), utilisant le protocole UDP. Ce mode nécessite l'adresse de destination à chaque envoi, et aucun accusé de réception n'est donné.

3.1.2 Sockets & Modèle OSI

Vous devez déjà avoir vu ce qu'est le modèle OSI (Open Systems Interconnection) ... Nous ne ferons ici qu'une très brève présentation de ce modèle.

3.1.2.1 Modèle OSI : le Modèle OSI a été conçu dans les années 1970 et devient une norme en 1977 portant la référence ISO 7498. L'objectif de cette norme est de spécifier un cadre général pour la création de normes ultérieures cohérentes. Le modèle comporte sept couches, numérotées de 1 à 7 :

1. La couche *physique* : est chargée de la transmission effective des signaux entre les interlocuteurs. Son service est limité à l'émission et la réception d'un bit ou d'un ensemble de bit sur le médium de communication.
2. La couche *liaison de données* gère les communications entre 2 machines directement connectées entre elles, ou connectées à un équipement.
3. La couche *réseau* gère les communications de proche en proche, généralement entre machines : routage et adressage des paquets. Dans ce niveau, on retrouve notamment le protocole *IP* (Internet Protocol).
4. La couche *transport* gère les communications de bout en bout entre processus. Les protocoles les plus utilisés dans ce niveau sont *TCP* (Transmission Control Protocol) et *UDP* (User Datagram Protocol)
5. La couche *session* gère la synchronisation des échanges et les "transactions", permet l'ouverture et la fermeture de session.
6. La couche *présentation* est chargée du codage des données applicatives, précisément de la conversion entre données manipulées au niveau applicatif et chaînes d'octets effectivement transmises.
7. La couche *application* est le point d'accès aux services réseaux, elle n'a pas de service propre spécifique dans la norme.

Ces couches sont généralement réparties en deux groupes :
- les quatre couches inférieures sont plutôt orientées communication et sont souvent fournies par un système d'exploitation.
- Les trois couches supérieures sont plutôt orientées application et réalisées par des bibliothèques ou un programme spécifique

Dans une telle architecture, coté émission, une couche de niveau $N+1$ envoie des données à l'entité de niveau N. Ces données seront encapsulées dans une enveloppe correspondant au protocole utilisé à ce niveau, et à leur tour envoyées au niveau $N-1$. Côté récepteur, chaque entité analyse l'enveloppe protocole correspondant à sa couche et transmet les données à la couche supérieure. Au dernier niveau, on doit retrouver la donnée émise.

Les sockets se situant juste au-dessus de la couche *transport* du modèle OSI, elles utilisent les protocoles des couches inférieures, TCP ou UDP et IP. Nous présentons brièvement ces protocoles ci-dessous.

3.1.2.2 Protocole IP : c'est un des protocoles les plus importants d'Internet car il permet l'élaboration et le transport des *datagrammes IP* (les paquets de données). Le protocole IP traite les datagrammes IP indépendamment les uns des autres en définissant leur représentation, leur routage et leur expédition. Ainsi, Le protocole IP détermine le destinataire d'un message à l'aide de 3 champs :
- Le champ adresse IP : adresse de la machine définie comme une suite de 4 octets séparés par un point (127.0.0.1 par exemple pour l'adresse locale d'une machine (ou adresse de bouclage))
- Le champ masque de sous-réseau qui permet au protocole IP de déterminer la partie de l'adresse IP qui concerne le réseau
- Le champ passerelle, permettant au protocole IP de savoir à quelle machine remettre le datagramme si jamais la machine de destination n'est pas sur le réseau local (information obtenue en combinant l'adresse de destination et le masque de sous-réseau).

3.1.2.3 Protocole TCP : c'est un des principaux protocoles de la couche transport. TCP est un protocole orienté connexion, c'est-à-dire qu'il permet à deux machines qui communiquent de contrôler l'état de la transmission. Le protocole TCP permet, entre autre, de :
- remettre en ordre les datagrammes en provenance du protocole IP,
- vérifier le flot de données afin d'éviter une saturation du réseau,
- formater les données en segments de longueur variable afin de les "remettre" au protocole IP,
- multiplexer les données, c'est-à-dire de faire circuler simultanément des informations provenant de sources (applications par exemple) différentes sur une même ligne,
- l'initialisation et la fermeture d'une communication de manière "courtoise"

Lors d'une communication utilisant le protocole TCP, les deux machines doivent établir une connexion. La machine émettrice (demandant la connexion) est appelée client, tandis que la machine réceptrice est appelée serveur. On se retrouve alors dans un environnement Client-Serveur. La propriété principale d'un tel environnement est que les entités communiquent en mode connecté, c'est-à-dire que la communication se fait dans les deux sens (comme un appel téléphonique).

3.1.2.4 Protocole UDP : le protocole UDP (User Datagram Protocol) est un protocole non orienté connexion de la couche transport. Le principe de ce protocole est d'assurer la transmission de données de manière relativement simple entre deux entités, chacune étant définie par une adresse IP et un numéro de port. Contrairement au protocole TCP, il fonctionne sans négociation : il n'existe donc pas de procédure de connexion et ne garantit pas la bonne livraison des datagrammes à destination, ni d'ailleurs leur ordre d'arrivée.

3.1.2.5 positionnement des sockets dans le modèle OSI comme représenté dans le tableau 1, les sockets prennent places au niveau des couches 5, 6 et 7, juste au-dessus de la couche transport, s'appuyant donc sur la couche transport (TCP ou UDP), puis la couche réseau (IP), puis les couches liaisons et physiques (Ethernet).

Modèle des sockets	Modèle OSI
Application utilisant les sockets	Application Présentation Session
UDP/TCP	Transport
IP	Réseau
Ethernet, X25, ...	Liaison Physique

TABLE 1 – Sockets et OSI.

3.2 Les sockets : mise en œuvre

Nous allons maintenant nous intéresser à la mise en œuvre des sockets en langage C et sous un environnement Unix/Linux [15]

3.2.1 Les structures de données

Dans un premier temps nous nous intéressons aux structures de données manipulées dans la mise en œuvre de sockets.

3.2.1.1 struct sockaddr_in : cette structure permet de fixer un certains nombre de paramètres concernant l'adresse internet de l'entité que l'on souhaite connecter. La structure contient différents champs explicité ci-dessous :

```
struct sockaddr_in {
  uint8_t         sin_len;      /* longueur totale      */
  sa_family_t     sin_family;   /* famille : AF_INET     */
  in_port_t       sin_port;     /* le numéro de port     */
  struct in_addr  sin_addr;     /* l'adresse internet    */
  unsigned char   sin_zero[8];  /* un champ de 8 zéros   */
};
```

Les champs sin_len et sin_zero ne sont pas utilisé par le programmeur. Nous nous intéresserons donc principalement aux champs sin_addr et sin_port, nous permettant de fixer adresse IP et numéro de port pour la communication.

3.2.1.2 struct sockaddr : comme vous le remarquerez dans les signatures des fonctions présentées dans la suite, le type de paramètre demandé par celles-ci pour la gestion des adresses est de type struct sockaddr. Cette structure est définie comme suit :

```
struct sockaddr {
  unsigned char   sa_len;       /* longueur totale          */
  sa_family_t     sa_family;    /* famille d'adresse  */
  char            sa_data[14];  /* valeur de l'adresse      */
};
```

15. La mise en œuvre sous Windows est un peu différente.

Elle est donc très proche de la structure struct sockaddr_in. Cette structure sert juste de référence générique pour les appels systèmes. Il vous faudra donc simplement "caster" votre sockaddr_in en sockaddr.

3.2.1.3 struct hostent : cette structure permet de faire le lien entre adresse et nom d'une entité. Elle permet donc, à l'aide des fonctions associées de manipuler des entités à un plus haut niveau d'abstraction. Cette structure se définit ainsi :

```
struct hostent {
  char    *h_name;        /* Nom officiel de l'hôte.    */
  char    **h_aliases;    /* Liste d'alias.             */
  int      h_addrtype;    /* Type d'adresse de l'hôte.  */
  int      h_length;      /* Longueur de l'adresse.     */
  char    **h_addr_list;  /* Liste d'adresses.          */
}
```

3.2.2 Les fonctions

Nous présentons ici les fonctions de bases permettant de mettre en place une communication en mode connecté ou déconnecté.

3.2.2.1 socket : int socket(int domain, int type, int protocol)
Cette fonction crée un socket. Un socket est donc de type int qui correspond au descripteur retourné par cette fonction. Concernant les paramètres :

- domain représente le type de protocole utilisé :
  - AF_INET pour TCP/IP utilisant une adresse Internet sur 4 octets
  - AF_UNIX pour les communications UNIX en local sur une même machine
- type indique le type de service (orienté connexion ou non). Dans le cas d'un service orienté connexion, c'est-à-dire utilisant le protocole TCP, ce paramètre doit prendre la valeur SOCK_STREAM (communication par flot de données).
  Dans le cas du communication en mode non connecté, c'est-à-dire utilisant le protocole UDP, ce paramètre doit prendre la valeur SOCK_DGRAM (utilisation de datagrammes, i.e. blocs de données)
- protocole permet de spécifier un protocole permettant de fournir le service désiré. Dans le contexte d'un service AF_INET il n'est pas utile, et on utilisera toujours 0 pour ce paramètre.

3.2.2.2 close : int close(int fd)
Cette fonction ferme le descripteur fd, qui correspond au descripteur de notre socket.

3.2.2.3 send et sendto : ces deux fonctions réalisent les même choses, mais dans des environnements différents. elles ont pour signature int send(int s, const void *msg, size_t len, int flags) et int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen).
send doit être utilisée pour le mode connecté. Elle envoie sur le socket s, les données pointées par msg, pour une taille de len octets.
sendto s'utilise elle pour le mode non connecté. le paramètre supplémentaire permet de spécifier notre structure sockaddr désignant le destinataire.
Ces 2 fonctions renvoient le nombre d'octets envoyés.

3.2.2.4 recv et recvfrom : comme précédemment, ce deux fonctions s'utilisent dans des contextes différents. Les signatures sont données ci-après : int recv(int s, void *buf, int len, unsigned int flags) et int recvfrom(int s, void *buf, int len, unsigned int flags struct sockaddr *from, socklen_t *fromlen).
recv s'utilise dans un contexte connecté. Elle reçoit sur le socket s, des données stockée dans le paramètre buf, pour une taille maximale de len octets.
recvfrom est à utiliser pour le mode non connecté. Là encore, il est nécessaire de spécifier une structure sockaddr.
Ces 2 fonctions retournent le nombre d'octets reçus.

3.2.2.5 bind : int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)
Cette fonction permet de lier un socket et une structure de type struct sockaddr.

3.2.2.6 connect : int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen)

Là, comme sont nom l'indique, la fonction permet de connecter le socket à l'adresse stockée dans le paramètre de structure struct sockaddr.
Il s'agit donc d'une fonction à utiliser coté du client.

3.2.2.7 listen : int listen(int s, int backlog)

Le rôle de cette fonction est de définir la taille s de la file d'attente de connexions pour un socket s.

3.2.2.8 accept : int accept(int sock, struct sockaddr *adresse, socklen_t *longueur)

Cette fonction autorise et accepte la connexion d'un socket sur le socket sock. Le socket doit préalablement être lié avec un port par l'utilisation de la fonction bind. Enfin, le paramètre adresse contiendra après exécution de cette fonction, les informations du client qui s'est connecté.
Cette fonction retourne un nouveau socket. C'est ce dernier qui devra être utilisé pour communiquer avec le client.

3.3 Exemple : simple client/serveur en mode TCP

Nous donnons ci-dessous le code commenté de l'implantation d'un (très) petit client/serveur utilisant un mode de communication connecté, i.e. TCP.
Le serveur attend une connexion, et lorsqu'un client se connecte, il envoie à ce client un message de bienvenue. D'autre part, il affiche l'adresse IP du client connecté.
Le client, lui, se connecte au serveur puis affiche le message de bienvenue reçu du serveur.

Nous créons deux programmes (projet), un pour le serveur et l'autre pour le client. Vous trouvez ci-dessous le code commenté du programme jouant le rôle de serveur.

```
/* Le SERVEUR */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORTNUM 2300

int main(int argc, char *argv[])
{
  char* msg = "Bonjour les gens !\n";

  struct sockaddr_in dest; /* Information de la machine cliente*/
  struct sockaddr_in serv; /* Information de la machine serveur*/
  int mysocket;            /* socket utilisé pour attendre une connexion */
  socklen_t socksize = sizeof(struct sockaddr_in);

  memset(&serv, 0, sizeof(serv));           /* initialisation de la structure avant de
      remplir les champs */
  serv.sin_family = AF_INET;                /* Fixe le type de socket réseau */
  serv.sin_addr.s_addr = htonl(INADDR_ANY); /* Fixe notre adresse IP (toutes les interfaces
      Physique) */
  serv.sin_port = htons(PORTNUM);           /* Fixe le numéro de port */

  mysocket = socket(AF_INET, SOCK_STREAM, 0); /*Création : la valeur SOCK_STREAM indique
      l'utilisation de TCP*/

  /* Association des infos serveur avec le socket */
  bind(mysocket, (struct sockaddr *)&serv, sizeof(struct sockaddr));

  /* Ecoute avec une taille de file d'attente de connexion de 1 */
  listen(mysocket, 1);

  /* accepte la connexion*/
  int consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);

  while(consocket)
```

```
  {
    printf("Connection entrante depuis %s - Envoi d'un message ...\n",
        inet_ntoa(dest.sin_addr));
    send(consocket, msg, strlen(msg), 0);

    /*fermeture de ce socket*/
    close(consocket);
    /*attente d'une nouvelle connexion*/
    consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);
  }

  close(mysocket);
  return EXIT_SUCCESS;
}
```

Voyons maintenant le code commenté du programme jouant le rôle de client :

```
/* LE CLIENT */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXRCVLEN 500
#define PORTNUM 2300

int main(int argc, char *argv[])
{
  char buffer[MAXRCVLEN + 1]; /* buffer pour la reception du message*/
  int len, mysocket;
  struct sockaddr_in dest;

  /* Création du socket */
  mysocket = socket(AF_INET, SOCK_STREAM, 0);

  memset(&dest, 0, sizeof(dest)); /* Initialisation de la structure*/
  dest.sin_family = AF_INET; /* Mode communication réseau */
  dest.sin_addr.s_addr = htonl(INADDR_LOOPBACK); /* Ip du serveur - localhost, 127.0.0.1*/
  dest.sin_port = htons(PORTNUM);                /* numéro de port utilisé */

  /* Connexion au serveur */
  connect(mysocket, (struct sockaddr *)&dest, sizeof(struct sockaddr));

  /* Reception du message en provenance du serveur */
  len = recv(mysocket, buffer, MAXRCVLEN, 0);

  /* Ajout du caractère de fin aux données reçues */
  buffer[len] = '\0';

  printf("Reception de %s (%d bytes).\n", buffer, len);

  /* Fermeture de la communication */
  close(mysocket);
  return EXIT_SUCCESS;
}
```

Exercice 8 : Repète Jacquot ! !

On souhaite réaliser une application client/serveur respectant le fonctionnement suivant : une fois connecté au serveur, le client demande un message à l'utilisateur (saisi via la console) et l'envoie au serveur. Le serveur après avoir reçu le message le renvoie au client. Ce dernier affiche le message reçu par le serveur sur la console.

1. *Étudier précisément l'exemple précédent.*
2. *Créer un projet C et implanter le coté serveur de cette application.*
3. *Créer un deuxième projet C et implanter le coté client de l'application.*
4. *Lancer le serveur, puis le client. Tester.*

4 Programmation multitâche

4.1 Thread & Processus

dans un système, un processus, est programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire. Durant son exécution, les instructions d'un programme modifient les valeurs des registres (compteur ordinal, registre d'état, ...) ainsi que le contenu de la pile. Il ne faut donc pas confondre programme et processus : le programme est la suite d'instructions (statique) exécuté par un processus : l'état de l'exécution d'un programme est défini par le contenu des registres et de la mémoire centrale (dynamique). Et les threads alors, c'est quoi ?

Les threads permettent de découper un programme en plusieurs tâches, celles-ci pouvant s'exécuter en pseudo parallèle ou en parallèle. Chaque tâche sera exécutée par un thread (porte aussi le nom de fil d'exécution, ou également de processus léger, ...).

Dans la majorité des systèmes d'exploitation actuels, un processus est au minimum composé d'un espace d'adressage et d'un thread de contrôle unique, le thread principal. Lors de l'exécution de vos programme, c'est ce dernier qui est en charge d'exécuter la fonction `main`. Un programme composé de plusieurs tâches vas donc, à l'exécution, donné lieu à un processus au sein duquel s'exécuteront plusieurs threads. Noter que tous les threads d'un même processus partagent le même espace d'adressage, et donc toutes les variables. Si ceci permet de simplifier la création d'un thread, il faudra porter une attention particulière au partage de données entre ceux-ci.

4.2 Thread Posix

Les threads POSIX, aussi appelés pthreads, sont un sous-standard de la norme POSIX décrivant une interface de programmation permettant de gérer des threads. Il s'agit du standard IEEE Std 1003.1c-1995 (POSIX.1c, Threads extensions). Cette interface est disponible sur la plupart des systèmes Unix modernes, comme Linux, les différentes variantes modernes de BSD, comme Mac OS X, mais n'est pas encore pas disponible nativement sous Windows [16].

Le standard définit un ensemble de types, de fonctions et de constantes en langage C. Le fichier à inclure dans les sources en C se nomme `pthread.h`. Une centaine de fonctions sont définies dans cette API. Ces fonctions permettent la création et la destruction de threads, l'ordonnancement des tâches, leur synchronisation et la gestion des données partagées.

4.2.1 Création & utilisation

Nous nous intéressons dans cette section à la création de thread Posix, nommé pthread dans la suite.

16. Il en existe plusieurs implémentations, dont une de Microsoft.

4.2.1.1 Type et fonctions : un pthread est modélisé par le type pthread_t. L'instruction ci-dessous permet donc de déclarer un pthread, action nécessaire avant sa création.

```
...
/*Declaration d'un pthread*/
pthread_t monPremierThread;
```

La création du pthread déclaré se fait alors par l'appel de l'instruction : int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg). Les paramètres de cette fonction sont définies comme suit :

- thread est initialisé par cette fonction, ce qui permettra d'identifier et de manipuler le pthread créé,
- attr permet de fixer certains paramètres du pthread créé ; l'utilisation d'une valeur NULL permet d'utiliser les caractéristiques par défaut,
- start routine est la fonction exécutée/invoquée par le pthread (c'est exactement ce qui se passe pour la fonction main et le thread principal) ;
- arg ce paramètre correspond au(x) paramètre(s) qui sera(seront) passé(s) paramètre à la fonction start_routine lors du lancement du pthread.
- le paramètre de retour, de type int, permet de tester la réussite de la création du pthread. La valeur retournée vaut 0 dans ce cas.

4.2.1.2 Lancement d'un pthread : là, ce n'est pas très compliqué ... le pthread est automatiquement lancée à la sortie de l'exécution de la fonction pthread_create, si cette dernière s'est terminée avec succès.

4.2.1.3 Fin d'un pthread : il est ici important de comprendre le lien entre le processus (exécution du programme) et le cycle de vie du pthread. En effet, lorsqu'un processus crée un pthread, s'il ne lui est pas explicitement indiqué d'attendre la fin d'exécution de ce pthread, alors à sa terminaison elle forcera l'arrêt de la tâche créée. L'API POSIX possède donc une fonction permettant d'attendre, si nécessaire, la fin d'un thread : il s'agit de la fonction int pthread_join(pthread_t *thread, void ** thread_return).

- thread est l'identifiant du pthread à attendre (correspond à celui initialisé par pthread_create)
- thread return, s'il est différent de NULL, recevra la valeur de retour du pthread (ou la valeur PTHREAD_CANCELED si le pthread a été annulé).

4.2.1.4 Endormir un pthread : il est souvent utile d'endormir un pthread lors d'opérations itératives. Il existe de nombreuses fonctionnalités pour faire cela, mais néanmoins, POSIX recommande l'utilisation de la fonction int nanosleep(const struct timespec *req, struct timespec *rem), dont les paramètres sont les suivants :

- req permet d'indiquer le temps d'endormissement souhaité par le passage d'une variable de type struct timespec contenant deux champs :
  - tv_sec : permet de fixer le nombre de secondes
  - tv_nsec : permet de fixer le nombre de nanosecondes

  Le temps demandé dans une variable tempscorrespondra donc à temps.tv_sec + temps.tv_nsec.
- rem utilisé lorsque la fonction nanosleep échoue. Dans ce cas, la variable passée en paramètre contiendra en sortie le temps restant lors de l'arrêt de la fonction.

L'exemple ci-dessous déclare et créé 2 pthread. Le pthread principal attend la fin de chacun d'eux pour se terminer. Les deux pthread exécutent la même fonction void * afficheUnMessage(void * mess) avec deux messages différents, chaque message étant passé en paramètre lors de la création du pthread. À chaque itération de la boucle for les deux pthread sont endormis durant $500ms$.

```
#include <pthread.h>
#include <cstdio>
#include <ctime>

/*Fonction exécutée par les pthreads*/
void * afficheUnMessage(void * mess){
  /*Déclaration d'une structure timespec*/
  /*0 seconde et 500 000 000 nanosecondes*/
  struct timespec tps, tps_rest;
  tps.tv_sec = 0;
  tps.tv_nsec = 500000000;
  int i;
  for(i=0;i<5;i++){
    printf("%s",(char *)mess);
   /*endort le pthread durant 500ms*/
   nanosleep(&tps,&tps_rest);
  }
```

```
return NULL;
}

/*Fonction permettant le test d'erreur*/
void messageErreur(int err, const char * mess){
  if (err       != 0){
    fprintf(stderr,"%s : %d\n",mess,err);
  }
}

int main(int argc, char **argv) {
  int erreur;
  /*Déclaration de 2 pthread*/
  pthread_t monPremierThread, monDeuxiemeThread;
  /*Création du pthread monPremierThread*/
  erreur = pthread_create(&monPremierThread,NULL,
  afficheUnMessage,(void *)"Bonjour depuis le thread 1\n");
  messageErreur(erreur,"Erreur créationThread");
  /*Création du pthread monDeuxiemeThread*/
  erreur = pthread_create(&monDeuxiemeThread,NULL,
  afficheUnMessage,(void *)"Bonjour depuis le thread 2\n");
  messageErreur(erreur,"Erreur créationThread");
  /*Attente de la terminaison du pthread monPremierThread*/
  erreur = pthread_join(monPremierThread,NULL);
  messageErreur(erreur,"Erreur Attente monPremierThread");
  /*Attente de la terminaison du pthread monDeuxiemeThread*/
  erreur = pthread_join(monDeuxiemeThread,NULL);
  messageErreur(erreur,"Erreur Attente monDeuxiemeThread");
  return 0;
}
```

4.2.1.5 Terminer un pthread : dans certains cas, il peut être nécessaire de terminer volontairement l'exécution d'un pthread. pour ce faire, POSIX prévoit l'utilisation de la fonction void pthread_exit(void *retval) ; avec comme paramètre :

— retval : le contenu de la variable passé en paramètre lors de l'appel de cette fonction sera disponible pour la fonction d'attente de terminaison d'un pthread pthread_join.

Exercice 9 : Mon premier thread

1. *Créer un pthread prenant une valeur entière en entrée, val et réalisant la multiplication* $val = val * (i + 1)$*, où* i *et le numéro d'itération. Le thread affiche sur la console le résultat de chaque opération et doit se terminer lorsque ce dernier est supérieur à 12000.*
2. *Le résultat est alors affiché par le thread principal.*

4.3 Synchronisation

La possibilité pour un pthread d'attendre la terminaison d'un autre pthread par l'utilisation de la fonction pthread join ne permettant dpas de couvrir l'ensemble des besoins, les thread POSIX proposent deux vrai mécanismes de synchronisation :

— les variables d'exclusion mutuelle qui sont des verrous permettant par exemple d'assurer l'accès d'un pthread à une variable,
— les variables de condition qui permettant de modéliser la notion d'évènements tels que la libération d'une ressource par exemple.

4.3.1 Variable d'exclusion mutuelle

4.3.1.1 Principe : une variable d'exclusion mutuelle (portant aussi le nom de verrou) est un mécanisme permettant de garantir/protéger l'accès unique à une ressource partagée entre plusieurs pthreads (données, ressources critiques, ...). Le type POSIX d'une telle variable est pthread_mutex_t.

Avant sa première utilisation, une telle variable doit être initialisée. Tout accès à la ressource protégée par ce mécanisme doit donc ensuite être précédé d'un verrouillage de la variable d'exclusion mutuelle. L'accès à la variable partagée étant terminé, il faut le faire suivre d'un déverrouillage de la variable d'exclusion mutuelle.

4.3.1.2 Initialisation & destruction : la fonction POSIX permettant de créer et d'initialiser une telle variable est définie comme suit : int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *mutexattr), où les paramètres correspondent à :

- mutex est la variable pthread_mutex_t à initialiser,
- mutexattr est le paramètre permettant de configurer le mutex ; l'utilisation de la valeur NULL pour ce paramètre provoque un comportement par défaut généralement satisfaisant dans la majorité des cas.

L'initialisation d'une variable d'exclusion mutuelle, ou mutex, se fera donc généralement ainsi :

```
...
/*Déclaration*/
pthread_mutex_t monMutex;
...
/*Initialisation*/
pthread_mutex_init(&(monMutex),NULL);
```

Lorsque la ressource n'existe plus, ou ne sera plus utilisée, il faut veiller à détruire la variable d'exclusion mutuelle associée à celle-ci. Pour ce faire, on utilise la fonction POSIX : int pthread_mutex_destroy(pthread_mutex_t *mutex). Le seul paramètre de cette fonction correspond à la variable d'exclusion mutuelle à détruire. Cette action pourra donc s'écrire :

```
...
/*Destruction de la variable d'exclusion mutuelle*/
pthread_mutex_destroy(monMutex);
...
```

4.3.1.3 Verrouillage & Déverrouillage : pour obtenir l'accès à une ressource en exclusion mutuelle, un pthread doit utiliser la fonction int pthread_mutex_lock(pthread_mutex_t *mutex). Le seul paramètre de cette fonction, mutex, est la variable d'exclusion mutuelle que l'on souhaite verrouiller.

Si la variable d'exclusion mutuelle n'est pas déjà verrouillée (par un autre pthread ou par celui exécutant la fonction pthread mutex lock()), la fonction se termine immédiatement et le pthread possède alors l'accès unique à la ressource protégée par le verrou obtenu, et ceci jusqu'à ce qu'il le libère explicitement.

Si un autre pthread a déjà verrouillé l'accès sur la variable d'exclusion mutuelle, alors le pthread est bloqué jusqu'à ce que le pthread détenteur du verrou le libère. Noter que si la ressource était déjà verrouillée par le pthread faisant sa demande, une erreur est généralement retournée.

La libération d'une variable d'exclusion mutuelle se fait par l'appel de la fonction int pthread_mutex_unlock(pthread_mutex_ *mutex) à condition que le pthread appelant possède déjà ce verrou. Le paramètre mutex correspond à la variable d'exclusion mutuelle à déverrouiller

Attention, il est généralement de la responsabilité du programmeur de faire en sorte qu'un verrou ne soit pas libéré par un pthread qui ne le possèderait pas.

4.3.1.4 Module de données : un module de données permet de réaliser une communication asynchrone entre deux tâches. L'accès à ce module doit bien évidemment être protégé afin d'éviter les erreurs de fonctionnement dues au problème de lecture/écriture. En utilisant la notion de variable d'exclusion mutuelle vue précédemment, il est alors possible de mettre en œuvre de tels modules.

L'exemple ci-dessous présente un module C réalisant l'implantation d'un module de donnée permettant le partage d'une valeur entière.

- monMDD.h :

```
#ifndef MDD_MONMDD_H_
#define MDD_MONMDD_H_

#include <pthread.h>

typedef struct {
```

```
  int val;
  pthread_mutex_t mutex;
}monMdd_t;

void initMdd(monMdd_t *mdd);

int lireValeur(monMdd_t *mdd);

void ecrireValeur(monMdd_t * mdd, int newval);

void detruireMdd(monMdd_t *mdd);

#endif /* MDD_MONMDD_H_ */
```

— monMDD.c :

```
#include "MonMdd.h"

void initMdd(monMdd_t *mdd){
  pthread_mutex_init(&(mdd->mutex),NULL);
}

int lireValeur(monMdd_t *mdd){
  pthread_mutex_lock(&(mdd->mutex));
  int v = mdd->val;
  pthread_mutex_unlock(&(mdd->mutex));
  return v;
}

void ecrireValeur(monMdd_t *mdd, int newval){
  pthread_mutex_lock(&(mdd->mutex));
  mdd->val = newval;
  pthread_mutex_unlock(&(mdd->mutex));
}

void detruireMdd(monMdd_t *mdd){
  pthread_mutex_destroy(mdd);
}
```

4.3.2 Variable condition

Dans cette section, nous présentons la notion de variable condition, permettant de modéliser l'attente ou l'arrivée d'un évènement. EN résumé, une fois qu'une telle variable a été créée, un pthread peut attendre que cette condition se réalise (i.e. attente passive de la condition), ou bien signaler que cette condition s'est réalisée. Cette notion est particulièrement utile pour résoudre un certain nombre de problèmes (comme le producteur/consommateur). Elle nous sera surtout utile par la suite pour mettre en œuvre la notion de tâches périodiques.

4.3.2.1 Initialisation & Destruction : une variable condition est initialisée en faisant appel à la fonction POSIX int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr) dans laquelle :

— cond est la variable condition à initialiser
— cond_attr est un paramètre permettant de configurer la condition.
La valeur NULL provoque un comportement par défaut utilisable dans la majorité des cas.

Une variable condition, lorsqu'elle n'est plus utilisée, peut être détruite par un appel à la fonction int pthread_cond_destroy(pt *cond) ; prenant en unique paramètre la variable condition à détruire.

4.3.2.2 Attente & Signal : un pthread peut donc se mettre en attente de la vérification d'une condition en utilisant la fonction int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex) avec comme paramètres :

— cond la condition de type pthread_cond_t à attendre ;
— mutex est une variable d'exclusion mutuelle que le pthread doit avoir préalablement verrouillée.

En ce qui concerne le fonctionnement de cette fonction, si la condition n'est pas vérifiée, le verrou mutex est relâché, et le pthread est mis en attente de réalisation de la condition. Une fois la condition réalisée, ce changement d'état est signalé et le verrou est à nouveau verrouillé par le pthread avant de terminer l'attente. Le pthread conserve donc l'exclusion mutuelle à son réveil.

Une des méthodes proposée par l'API POSIX pour signaler la réalisation d'une condition est la fonction int pthread_cond_signal(pthread_cond_t *cond).

Un exemple de mise en œuvre de ces variables condition est donné dans la section suivante, permettant l'implantation d'une tâche périodique.

4.4 pthread périodiques

Dans une application multitâche, certaines de ces tâches doivent s'exécuter de manière périodique. Plus précisément, une tâche τ_1 disposant d'une période de T unités de temps doit toujours débuter une "nouvelle exécution" après l'écoulement de ce temps. Dans certaines applications (boucle de commande, ...), cette périodicité doit être très stricte et rigoureuse. Dans ce cas, l'application doit s'exécuter en respectant parfaitement ce critère.
Dans sections précédentes, nous avons utilisé la fonction nanosleep pour endormir (i.e. faire attendre) une tâche pendant une période T. Mais cette technique n'est pas très rigoureuse vis-à-vis de la périodicité de la tâche. En effet, il se produit une dérive d'horloge due au fait que le temps d'attente se rajoute au temps d'exécution des instructions contenues dans le pthread.
Pour palier ce phénomène de dérive d'horloge, il est nécessaire d'avoir un référentiel permettant de connaitre les instants précis de réveil de la tâche. Ainsi, nous présentons ci-dessous deux techniques utilisant directement le temps fourni par l'horloge système pour implanter des tâches strictement périodiques.

4.5 Solution utilisant une variable condition

Pour récupérer le temps de l'horloge du système il faut faire appel à la fonction int clock_gettime(clockid_t clk_id, struct timespec *time). Les paramètres de cette fonction correspondent à :

- TIME une variable de type timespec dans laquelle la valeur de la période T est fixée,
- clk_id est un paramètre pouvant prendre les valeurs CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID ou CLOCK_THREAD_CPUTIME_ID

En supplément de cette fonction, cette solution utilise plusieurs notions présentées précédemment et plus particulièrement :

- la structure de donnée struct timespec,
- une variable d'exclusion mutuelle,
- et une variable condition.

Enfin, pour endormir le pthread jusqu'à la date de sa prochaine exécution, nous utiliserons la fonction int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *verrou, struct timespec *time). Cette dernière utilise un timeout sur le temps time pour réveiller la tâche endormie sur l'attente de la variable condition cond (qui ne sera jamais signalée ... sinon ça ne fonctionnerait pas ...).

L'exemple ci dessous montre l'implantation d'un pthread strictement périodique, de période $T = 2s$, s'exécutant 10 fois.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

void *duTravail(struct timespec *periode){
  pthread_cond_t cond;
  pthread_mutex_t verrou;
  struct timespec temps;

  pthread_cond_init(&cond, NULL);
  pthread_mutex_init(&verrou, NULL);

  int i=0;
  /*date de départ*/
  clock_gettime(CLOCK_REALTIME, &temps);
  while(i<10){
    pthread_mutex_lock(&verrou);
    /*prochaine date*/
    temps.tv_sec = temps.tv_sec + periode->tv_sec;
    printf("La tache %s s'execute à l'instant %d secondes\n",
    "t1", (int)temps.tv_sec);
    i++;
    /*Bloqué jusqu'à la prochaine date*/
```

```
    pthread_cond_timedwait(&cond, &verrou, &temps);
    pthread_mutex_unlock(&verrou);
  }
  return NULL;
}

int main(int argc, char **argv) {
  pthread_t tache1;
  /*On fixe la période*/
  struct timespec periode;
  periode.tv_sec=2;
  periode.tv_nsec=0;
  /*Création du Thread*/
  pthread_create(&tache1, NULL,(void *) duTravail, (void*) &periode);
  /*Attente fin Thread*/
  pthread_join(tache1, NULL);
  return 0;
}
```

4.6 Solution utilisant clock_nanosleep

Comme nanosleep, clock_nanosleep permet au pthread appelant de s'endormir pendant une durée indiquée avec une précision de la nanoseconde. Il diffère de la fonction précédente dans le fait qu'il permet de choisir l'horloge avec laquelle la durée du sommeil sera mesurée et d'indiquer une valeur absolue ou relative pour cette durée. La signature de cette fonction est : int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request, struct timespec *remain). Le paramètre clock_id précise l'horloge avec laquelle sera mesurée la durée du sommeil. Il doit prendre l'une des valeurs suivantes :

- CLOCK_REALTIME : horloge temps réel configurable à l'échelle du système.
- CLOCK_MONOTONIC : horloge monotonique, non configurable, mesurant le temps depuis un instant du passé non spécifié qui ne change pas après le démarrage du système. *Nous choisirons ce deuxième paramètre.*
- CLOCK_PROCESS_CPUTIME_ID : horloge configurable par processus mesurant le temps processeur consommé par tous les thread du processus.

Le paramètre flags peut lui aussi prendre différentes valeurs.

- Si flags vaut 0, la valeur indiquée dans request est interprétée comme une durée relative à la valeur actuelle de l'horloge choisie via le paramètre clock_id.
- Si flags vaut TIMER_ABSTIME, le paramètre request est interprété comme un temps absolu tel qu'il est mesuré par l'horloge choicie via clock_id. Attention, si request est inférieur ou égal à la valeur de l'horloge, la fonction clock_nanosleep termine immédiatement sans suspendre le thread appelant.

Les deux autres paramètres correspondent à ceux explicités lors de la présentation de la fonction nanosleep.

L'exemple ci-dessous reprend le principe de l'exemple précédent, mais en utilisant cette deuxième solution permettant d'implanter une tâche strictement périodique.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

void *duTravail(struct timespec *periode){
  struct timespec temps, temps_rest;
  int i=0;
  while(i<10){
    clock_gettime(CLOCK_MONOTONIC, &temps);
    printf("La tache %s s'execute à l'instant %d secondes\n","t1", (int)temps.tv_sec);
    i++;
    //Gestion de la périodicité
    temps.tv_sec = temps.tv_sec + periode->tv_sec;
    temps.tv_nsec = temps.tv_nsec + periode->tv_nsec;
    if(temps.tv_nsec >= 1000000000) {
      temps.tv_nsec -= 1000000000;
      temps.tv_sec++;
    }
    clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME,&temps,&temps_rest);
  }
  return NULL;
}
```

```
int main(int argc, char **argv) {
  pthread_t tache1;
  struct timespec periode;
  periode.tv_sec=2;
  periode.tv_nsec=500000000;
  pthread_create(&tache1, NULL,(void *) duTravail, (void*) &periode);
  pthread_join(tache1, NULL);
  return 0;
}
```

Exercice 10 : Tâches périodiques

On souhaite réaliser une application C++ dans laquelle s'exécute deux tâches :

- *la première écrit périodiquement une valeur (correspondant à son nombre d'itérations) avec une période de* $1,5s$
- *la deuxième récupère périodiquement la valeur mise à disposition par la tâche précédente avec un rythme de* $750ms$

Ces deux tâches s'exécutent de manière infinie.

1. *Créer un module permettant de partager l'information entre les deux tâches.*
2. *Créer le premier pthread implémentant la première tâche.*
3. *Créer le deuxième pthread implémentant la deuxième tâche.*
4. *Créer le thread principal responsable du lancement de ces deux tâches.*

5 Réalisation du projet

Nous allons maintenant rentrer dans le vif du sujet, la réalisation de notre projet !

5.1 Préambule

5.1.1 Liaison descendante

Avant de découper le projet en différents modules, nous allons commencer par deux petits exercices permettant de mettre en œuvre différentes notions utiles quelque soit le module que vous choisirez de réaliser.

Le premier exercice permet de visiter les notions de communication MAVLink (format et manipulation d'un message de ce protocole) et de programmation réseaux (connexion au simulateur et lecture d'une trame en provenance de ce dernier) pour une communication descendante entre le simulateur et votre programme.

⋆ Exercice 11 ⋆ : Ecoute du simulateur - V1

Après avoir consciencieusement lu la section concernant la programmation réseau et avoir pris connaissance des informations concernant les messages MAVLink[a] *récupérer le fichier* ecoutesimu1.c.

1. *Compléter les parties manquantes concernant la connexion au simulateur.*
2. *Compléter les parties manquantes concernant l'accès aux informations d'un message MAVLink.*
3. *Compiler votre fichier en utilisant la commande :* gcc -o ecoutesimu1.out ecoutesimu1.c -I/home/avio2016/Documents/ArduPilot/ ardupilot/libraries/GCS_MAVLink/include/common/.
4. *Lancer le simulateur. Lancer votre programme.*
5. *Que remarquez vous ?*

a. Cf. section correspondante sur le moodle.

Le deuxième exercice, qui fait suite au précédent, permet de mettre en œuvre les notions de bases de la programmation multi-tâches (multi-threads).

⋆ Exercice 12 ⋆ : Ecoute du simulateur - V2

Après avoir consciencieusement lu la section concernant la programmation multi-threads, réaliser les questions ci-dessous.

1. *Dupliquer le fichier de code précédent dans un nouveau fichier.*
2. *Modifier la fonction* main *de ce fichier de telle sorte à en faire une fonction classique (i.e. un sous-programme).*
3. *Implémenter une nouvelle fonction* Main *dans laquelle sera déclarer un thread s'appuyant sur la fonction précédente et qui réalisera le lancement (i.e. la mise en exécution) de ce dernier.*

5.1.2 Liaison montante

Vous disposez maintenant de la majorité des notions pour réaliser l'exercice qui suit. Il s'agit ici de réaliser un programme permettant de changer le mode du simulateur via différentes touches du clavier.

⋆ Exercice 13 ⋆ : Commande du simulateur

Réaliser un programme permettant de changer le mode du simulateur à la suite d'une frappe sur une touche du clavier. Pour cela vous devrez :

1. *Initier une communication montante avec le simulateur (inspirez vous des programmes précédents ...)*
2. *mettre le programme ne attente d'un événement clavier*
3. *Construire et envoyer la trame Mavlink correspondant à la touche du clavier*