

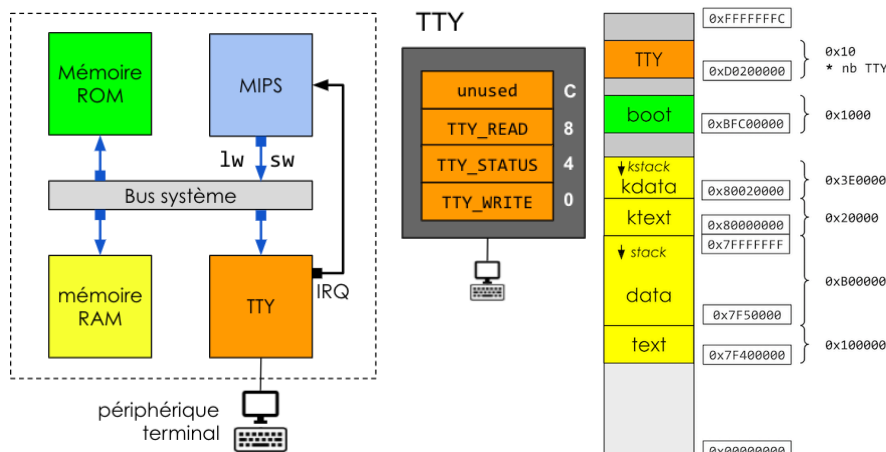
# Boot et premier programme en mode kernel

## 1. Analyse de l'architecture

1. Analyse de l'architecture
2. Programmation assembleur
3. Compilation et édition de liens
4. Programmation en C
6. Usage de Make (optionnel)

Les trois figures ci-dessous donnent des informations sur l'architecture du prototype **almo1** sur lequel vous allez travailler.

- À gauche, vous avez un schéma simplifié.
- Au centre, vous avez la représentation des 4 registres internes du contrôleur de terminal **TTY** nécessaires pour commander un couple écran-clavier.
- À droite, vous avez la représentation de l'espace d'adressage du prototype.



## Questions

- Il y a deux mémoires dans **almo1** : RAM et ROM. Qu'est-ce qui les distinguent et que contiennent-elles ? (C9 S6+S9)
  - La ROM est une mémoire morte, c'est-à-dire en lecture seule. Elle contient le code de démarrage du prototype.
  - La RAM est une mémoire vive, c'est-à-dire pouvant être lue et écrite. Elle contient le code et les données.
- Qu'est-ce l'espace d'adressage du MIPS ? Quelle taille fait-il ? Quelles sont les instructions du MIPS permettant d'utiliser ces adresses ? Est-ce synonyme de mémoire ? (C9 S7)
  - L'espace d'adressage du MIPS est l'ensemble des adresses que peut former le MIPS.
  - Les adresses sont sur 32 bits et désignent chacune un octet, il y a donc  $2^{32}$  octets.
  - On accède à l'espace d'adressage avec les instructions load/store (**lw**, **lh**, **lb**, **lhu**, **lbu**, **sw**, **sh**, **sb**).
  - Non, les mémoires sont des composants contenant des cases de mémoire adressable. Les mémoires sont placées (on dit aussi « *mappées* » dans l'espace d'adressage).
- Dans quel composant matériel se trouve le code de démarrage et à quelle adresse est-il placé dans l'espace d'adressage et pourquoi à cette adresse ? (C9 S6+S7)
  - Le code de boot est dans la mémoire ROM.
  - Il commence à l'adresse **0xBFC00000** parce que c'est l'adresse qu'envoie le MIPS au démarrage.
- Quel composant permet de faire des entrées-sorties dans **almo1** ? Citez d'autres composants qui pourraient être présents dans un autre SoC ? (C9 S6+connaissances personnelles)
  - Ici, c'est le composant **TTY** qui permet de sortir des caractères sur un écran et de lire des caractères depuis un clavier.
  - Dans un autre SoC, on pourrait avoir un contrôleur de disque, un contrôleur vidéo, un port réseau Ethernet, un port USB, des entrées analogiques (pour mesurer des tensions), etc.
- Il y a 4 registres de commande dans le contrôleur de **TTY**, à quelles adresses sont-ils placés dans l'espace d'adressage ? Comme ce sont des registres, est-ce que le MIPS peut les utiliser comme opérandes pour ses instructions (comme add, or, etc.) ? Dans quel registre faut-il écrire pour envoyer un caractère sur l'écran du terminal (implicitement à la position du curseur) ? Que contiennent les registres **TTY\_STATUS** et **TTY\_READ** ? Quelle est l'adresse de **TTY\_WRITE** dans l'espace d'adressage ? (C9 S10)
  - Le composant **TTY** est placé à partir de l'adresse **0xD0200000**.
  - Non, ce sont des registres de périphériques placés dans l'espace d'adressage et donc accessibles par des instructions load/store uniquement.
  - Pour écrire un caractère sur l'écran, il faut écrire le code ASCII du caractère dans le registre **TTY\_WRITE**.
  - TTY\_STATUS** contient 1 s'il y a au moins un caractère en attente d'être lu, **TTY\_READ** contient le code ASCII du caractère tapé au clavier si **TTY\_STATUS==1**.
- Le contrôleur de **TTY** peut contrôler de 1 à 4 terminaux. Chaque terminal dispose d'un ensemble de 4 registres (on appelle ça une carte de registres, ou en anglais une *register map*). Ces ensembles de 4 registres sont placés à des adresses contiguës. S'il y a 2 terminaux (**TTY0** et **TTY1**), à quelle adresse est le registre **TTY\_READ** de **TTY1** ? (C9 S10)
  - Si les adresses utilisées par **TTY0** commencent à **0xd0200000** alors celles de **TTY1** commencent à l'adresse **0xd0200010** et donc **TTY\_READ** est à l'adresse **0xd0200018**.
- Que représentent les flèches bleues sur le schéma ci-dessus ? Pourquoi ne vont-elles que dans une seule direction ? (C9 S11)

- Ces flèches représentent le sens des requêtes d'accès à la mémoire, c'est-à-dire les *loads* et les *stores*. Ces requêtes sont émises par le MIPS lors de l'exécution des instructions `lw`, `sw`, etc. Les requêtes sont émises par le MIPS et reçues par les composants mémoires ou les périphériques.
- On ne représente pas les données qui circulent, mais juste les requêtes, pour ne pas alourdir inutilement le schéma. Implicitement, si le MIPS envoie une requête de lecture alors il y aura une donnée qui va revenir, c'est obligatoire, alors on ne la dessine pas, car ce n'est pas intéressant. En revanche, le fait que le MIPS soit le seul composant à émettre des requêtes est une information intéressante.

## 2. Programmation assembleur

L'usage du code assembleur est réduit au minimum. Il est utilisé uniquement où c'est indispensable. C'est le cas du code de démarrage. Ce code ne peut pas être écrit en C pour au moins la raison suivante : le compilateur C suppose la présence d'une pile et d'un registre du processeur contenant le pointeur de pile, or au démarrage le contenu des registres n'est pas significatif. Dans cette partie, nous allons nous intéresser à quelques éléments de l'assembleur qui vous permettront de comprendre le code en TP.

### Questions

1. Nous savons que l'adresse du premier registre du `TTY` est `0xd0200000` est qu'à cette adresse se trouve le registre `TTY_WRITE` du `TTY0`. (C9 S10)  
Écrivez le code permettant d'écrire le code ASCII `'x'` sur le terminal 0. Vous avez droit à tous les registres du MIPS puisqu'à ce stade il n'y pas de conventions sur leur utilisation.  
Ce qu'il faut bien comprendre, c'est que l'adresse du registre `TTY_WRITE` est l'adresse d'une sortie du SoC, ce n'est pas une mémoire à proprement parler. Il est d'ailleurs interdit de lire à cette adresse. Pour écrire un message à l'écran, il faut écrire tous les caractères du message à cette adresse (`0xd0200000`).

```
lui    $4, 0xd020    // $4 ← 0xd0200000
ori    $5, $0, 'x'    // $5 ← code ascii de x
sb     $5, 0($4)      // TTY0.TTY_WRITE ← x
```

En principe, entre chaque écriture, il faut attendre un peu que le caractère précédent soit parti, parce que le débit du port de sortie matériel (USB par exemple) est beaucoup plus lent que ce que peut faire le processeur. Dans notre cas, c'est un simulateur de SoC et les caractères sont envoyés vers un terminal sans délai. Dans ce cas, il n'est pas nécessaire d'attendre.

2. Dans la question précédente, on connaissait l'adresse absolue du registre `TTY_WRITE`. On suppose désormais que l'adresse du premier registre du `TTY` se nomme `__tty_regs_map` (c'est le nom du label). Le programmeur ne connaît pas l'adresse, il ne connaît que le nom du label (on dit aussi symbole). Ainsi, pour écrire `'x'` sur le terminal 0, nous devons utiliser la macro instruction `la $r, label`. Cette macro-instruction est remplacée lors de l'assemblage du code par les instructions `lui` et `ori`. Il existe aussi la macro instruction `li` qui demande de charger une valeur sur 32 bits dans un registre.

Pour être plus précis, les macro-instructions

```
la $r, label
li $r, 0x87654321
```

sont remplacées par

```
lui $r, label>>16    // chargement des 16 bits de poids forts de label
ori $r, $r, label & 0xFFFF // chargement des 16 bits de poids faible de label
lui $r, 0x8765        // chargement des 16 bits de poids fort de 0x8765431
ori $r, $r, 0x4321    // chargement des 16 bits de poids faible de 0x8765431
```

Réécrivez le code de la question précédente en utilisant `la` et `li` (C9 S34)

- Il suffit de remplacer les instructions `lui` et `ori` par `la` et `li`.

```
la    $4, __tty_regs_map
li    $5, 'x'
sb     $5, 0($4)
```

3. En assembleur pour sauter à une adresse de manière inconditionnelle, on utilise les instructions `j label` et `jr $r`. Ces instructions permettent-elles d'effectuer un saut à n'importe quelle adresse ?

Attention, la réponse n'est pas dans le cours, mais dans la connaissance du codage des instructions de saut (`jump` et `branch`) que vous avez vu au début du module.

- Il faut avoir compris que l'instruction `j` et toutes les branchements (`bne`, `beq`, etc.) sont relatives au PC. Il n'est pas possible d'aller n'importe où dans l'espace d'adressage.
- `j label` effectue un saut relativement au `PC` puisque le `label` n'est pas entièrement encodé dans l'instruction binaire.  
Cette instruction réalise :  
 $PC \leftarrow (PC \& 0xf0000000) + (ZeroExtended(Imm26, 32) \ll 2)$   
Les 4 bits de poids forts du `PC` sont conservés, le saut est bien relatif au PC (`ZeroExtended` désigne ici le fait d'étendre le label sur 32 bits en ajoutant des zéros en tête).  
Autrement dit, si l'instruction `j label` est à l'adresse `PC`, l'adresse `label` doit avoir le même chiffre de poids fort (en hexadécimal), c'est-à-dire les mêmes 4 bits de poids fort (b31 à b28) en binaire, sinon l'assembleur provoque une erreur lors du codage.
- A l'inverse, `jr $r` effectue un saut absolu puisque cette instruction réalise  $PC \leftarrow $r$
- En conséquence, si l'on veut sauter n'importe où en mémoire, il faut utiliser l'instruction `jr`.

4. Vous avez utilisé les directives `.text` et `.data` pour définir les sections où placer les instructions et les variables globales, mais il existe la possibilité de demander la création d'une nouvelle section dans le code objet produit par le compilateur avec la directive `.section name,"flags"`

- `name` est le nom de la nouvelle section. On met souvent un `.name` (avec un `.` au début) pour montrer que c'est une section et
- `"flags"` informe sur le contenu : `"ax"` pour des instructions, `"ad"` pour des données (ceux que ça intéresse pourront regarder le manuel de l'assembleur `Assembleur/Directives/.section`)

Écrivez le code assembleur créant la section `".mytext"` et suivi de l'addition des registres `$5` et `$6` dans `$4` (C9 S10)

- Pour répondre, il faut avoir compris l'explication donnée dans la question. L'intérêt de cette question est de revenir sur la notion de section. Une section est un segment d'adresses ayant un but spécifique. On définit des sections pour le code, pour les différents types de données (initialisées, non initialisées, en lecture-seule, etc.). Ces sections sont placées dans l'espace d'adressage par l'éditeur de liens (`ld`) et la manière dont elles sont placées est définie dans un fichier nommé `ldscript` donné en paramètre de l'éditeur de lien.

```
.section .mytext,"ax"
mylabel:
addu $4,$5,$6
```

5. À quoi sert la directive `.globl label` dans un programme assembleur ? (C9 S34)

- `globl` signifie `glob al label`. Cette directive permet de dire que le `label` est visible en dehors de son fichier de définition. Ainsi il est utilisable dans les autres fichiers assembleur ou les autres fichiers C du programme.
- Ce qu'il faut comprendre, c'est que les comportements du C et de l'assembleur vis-à-vis des labels sont inversés. Dans un fichier C, quand on définit un label (une fonction ou variable), ce label est par défaut `extern`, c'est-à-dire qu'il est utilisable dans un autre fichier C. Si on veut que le label ne soit utilisable que dans le fichier dans lequel il est défini, il faut utiliser le mot clé `static` lors de sa déclaration. En assembleur, c'est l'inverse, les labels sont par défaut `static`, c'est-à-dire utilisable uniquement dans le fichier où ils sont définis. Si on veut qu'ils soient utilisables dans les autres fichiers, il faut le dire avec la directive `.globl`.

6. Écrivez une séquence de code qui affiche la chaîne de caractère `"Hello"` sur `TTY0`. Vous devez déclarer la chaîne de caractère `"Hello"` dans la section `.data`, puis écrire le code dans la section `.text`. Vous pouvez utiliser tous les registres que vous voulez. Vous supposez que le label `__tty_regs_maps` est déjà défini et qu'il désigne le premier registre de commande du `TTY0` qui est `TTY_WRITE`. C'est une boucle qui écrit chaque caractère de la chaîne dans le registre `TTY_WRITE` du `TTY0`, jusqu'à trouver le `0` de fin de chaîne.

- On pourrait choisir d'autres registres.

```
.data
hello: .asciiz "Hello"
.text
    la    $4, hello           // $4 ← address of string
    la    $5, __tty_regs_map  // $5 ← address of tty's registers map

print:
    lb    $8, 0($4)          // get current char
    sb    $8, 0($5)          // send the current char to the tty
    addiu $4, $4, 1          // point to the next char
    bne   $8, $0, print      // check that it is not null, if ok it must be printed
```

7. En regardant le dessin de l'espace d'adressage du prototype **almo1** (plus haut et sur le slide 7 du cours 9), dites à quelle adresse devra être initialisé le pointeur de pile **pour le kernel**. Rappelez pourquoi c'est indispensable de le définir avant d'appeler une fonction C et écrivez le code qui fait l'initialisation, en supposant que l'adresse du pointeur de pile a pour nom `__kdata_end`.

- On parle ici du pointeur de pile pour le `kernel`, on doit mettre le pointeur de pile dans la section des données du kernel. Ici, il n'y en a qu'une et c'est `.kdata`. La pile est mise en haut de cette section aux adresses les plus grandes. Les adresses du bas de la section sont occupées par les données globales du kernel.
- La pile va être initialisée juste à la première adresse au-delà de la zone `kdata` donc `__kdata_end = 0x80020000 + 0x003E0000 = 0x80400000`. Oui, on initialise le pointeur de pile à une adresse qui est en dehors du segment! Ce n'est pas un problème parce que le compilateur C n'écrira jamais à cette adresse, car elle n'est pas dans le contexte de la fonction C et que la fonction appelée ici n'a pas d'argument `void kinit(void)`. La première chose que la fonction `kinit()` fait, c'est décrémenter le pointeur de pile pour allouer son contexte. Si la fonction `kinit()` avait eu des arguments, alors il aurait fallut prévoir de l'espace dans la pile pour ces arguments.
- En effet, la première chose que fait une fonction, c'est décrémenter le pointeur de pile pour écrire `$31`, etc. Il faut donc que le pointeur ait été défini avant d'entrer dans la fonction.

```
la $29, __kdata_end
```

### 3. Compilation et édition de liens

Pour obtenir le programme exécutable, nous allons utiliser :

- `gcc -o file.o -c file.c`
  - Appel du compilateur avec l'option `-c` qui demande à `gcc` de faire la *préprocessing* puis la compilation C pour produire le fichier objet `file.o`.
- `ld -o bin.x -Tkernel.ld files.o ...`
  - Appel de l'éditeur de liens pour produire l'exécutable `bin.x` en assemblant tous les fichiers objets `.o`, en les plaçant dans l'espace d'adressage et résolvant les liens entre eux. Autrement dit, quand un fichier `f1.o` utilise une fonction `funf2()` ou une variable `varf2` définie dans un autre fichier `f2.o`, alors l'éditeur de liens place dans l'espace d'adressage les sections `.text` et `.data` des fichiers `f1.o` et `f2.o`, puis il détermine alors quelles sont les adresses de `funf2()` et `varf2` dans l'espace d'adressage et il complètent les instructions de `f1` qui utilisent ces adresses.
- `objdump -D file.o > file.o.s` ou `objdump -D bin.x > bin.x.s`
  - Appel du désassembleur qui prend les fichiers binaires (`.o` ou `.x`) pour retrouver le code produit par le compilateur pour le debug.

#### Questions sur l'édition de lien

Le fichier `kernel.ld` décrit l'espace d'adressage et la manière de remplir les sections dans le programme exécutable. Ce fichier est utilisé par l'éditeur de lien. C'est un `ldscript`, c'est-à-dire un `script` pour `ld`.

```
__tty_regs_map = [... question 1 ...] ;
__boot_origin = 0xbfc00000 ;
__boot_length = 0x00001000 ;
```

```

__ktext_origin   = 0x80000000 ;
__ktext_length   = 0x00020000 ;
[... question 1 ...]
__kdata_end      = __kdata_origin + __kdata_length ;

MEMORY {
    boot_region : ORIGIN = __boot_origin, LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
[... question 2 ...]
}

SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
[... question 3 ...]
    .kdata : {
        *(.*data*)
    } > kdata_region
}

```

1. Le fichier `kernel.ld` commence par la déclaration des variables donnant des informations sur les adresses et les tailles des régions de mémoire. Ces symboles n'ont pas de type et ils sont visibles de tous les programmes C. En regardant dans le dessin de la représentation de l'espace d'adressage, complétez les lignes de déclaration des variables pour la région `kdata_region`. Pour répondre, il faut savoir interpréter le dessin représentant l'espace d'adressage. (C9 S7+S38+S39)

- Si ça semble difficile, il faut revoir ce qu'est l'espace d'adressage.

```

__tty_regs_map   = 0xD0200000 ;
__kdata_origin   = 0x80020000 ;
__kdata_length   = 0x003E0000 ;

```

2. Le fichier contient ensuite la déclaration des régions (dans `MEMORY{...}`), c'est-à-dire les segments d'adresse en mémoire qui seront remplies par l'éditeur de lien avec les sections trouvées dans les fichiers objets selon un ordre décrit dans la partie `SECTIONS{}` du `ldscript`. Complétez cette partie (la zone `[... question 2 ...]`) pour ajouter les lignes correspondant à la déclaration de la région `kdata_region` ? (C9 S38+S39)

- La syntaxe est assez explicite, cela ne devrait pas poser de problème.

```

kdata_region : ORIGIN = __kdata_origin, LENGTH = __kdata_length

```

3. Enfin le fichier décrit comment sont remplies les régions avec les sections. Complétez les lignes correspondant à la description du remplissage de la région `ktext_region`. Vous devez la remplir avec les sections `.text` issus de tous les fichiers. Il faut bien comprendre que `.ktext` est une section produite par l'éditeur de liens. C'est ce que l'on appelle une section de sortie. `.text` est une section que l'éditeur de liens trouve dans un fichier objet `.o`, c'est ce que l'on appelle une section d'entrée. Comme il y a plusieurs fichiers objet, on doit dire à l'éditeur de lien de prendre toutes les sections `.text` de tous les fichiers qu'on lui donne. (C9 S38+S39)

- Le `*` devant `(.text)` est une expression régulière permettant de dire à l'éditeur de liens quels fichiers sont concernés, ici avec `*` c'est tous les fichiers. Les expressions régulières sont celles qu'on utilise avec le `shell`.

```

.ktext : {
    *(.text)
} > ktext_region

```

## 4. Programmation en C

Vous savez déjà programmer en C, mais vous allez voir ici des syntaxes ou des cas d'usage que vous ne connaissez peut-être pas encore. Les questions qui sont posées ici n'ont pas toutes été vues en cours, mais vous connaissez peut-être les réponses, sinon ce sera l'occasion d'apprendre.

### Questions

1. Quels sont les usages du mot clé `static` en C ? (c'est une directive que l'on donne au compilateur C)

- Le cours 9 n'en parle pas, mais dans le code vous trouverez cette directive un peu partout. Il y a deux usages.
  1. Déclarer `static` une variable globale ou une fonction en faisant précéder leur définition du mot clé `static` permet de limiter la visibilité de cette variable ou de cette fonction au seul fichier de déclaration. Notez que par défaut les variables et les fonctions du C ne sont pas `static`, il faut le demander explicitement. C'est exactement l'inverse en assembleur où tout label est implicitement `static` ; il faut demander avec la directive `.globl` de le rendre visible.
  2. Déclarer `static` une variable locale (dans une fonction donc) permet de la rendre persistante, c'est-à-dire qu'elle conserve sa valeur entre deux appels. Cette variable locale n'est pas dans le contexte d'exécution de la fonction (c'est-à-dire qu'elle n'est pas dans la pile parce que le contexte d'exécution est libéré en sortie de fonction). Une variable locale `static` est en fait allouée comme une variable globale mais son usage est limité à la seule fonction où elle est définie.

2. Pourquoi déclarer des fonctions ou des variables `extern` ?

- Ça n'a plus ce n'est pas dit dans le cours mais c'est sensé être connu. Notez que la directive externe est implicite en C et qu'on peut donc ne pas l'écrire. On la met pour la lisibilité du code.
- Les déclarations `extern` permettent d'informer que le compilateur qu'une variable ou qu'une fonction existe et est définie dans un autre fichier. Le compilateur connaît ainsi le type de la variable ou du prototype des fonctions, il sait donc comment les utiliser. En C, par défaut, les variables et les fonctions doivent être déclarées / leur existence et leur type doivent être connus avant leur utilisation.
- Il n'y a pas de déclaration `extern` en assembleur parce que ce n'est pas un langage typé. Pour l'assembleur, un label c'est juste une adresse donc un nombre.

3. Comment déclarer un tableau de structures en variable globale ? La structure est nommée `test_s`, elle a deux champs `int` nommés `a` et `b`. Le tableau est nommé `tab` et a 2 cases.

- Là encore, ce sont des connaissances censées être connues, mais c'est important parce qu'on a besoin de le comprendre pour la déclaration des registres du TTY.

```
struct test_s {
    int a;
    int b;
};
struct test_s tab[2];
```

4. Supposons que la structure `tty_s` et le tableau de registres de `TTY` soient définis comme suit. Écrivez la fonction C `int getchar0(void)` bloquante qui attend un caractère tapé au clavier sur le `TTY0`. Nous vous rappelons qu'il faut attendre que le registre `TTY_STATUS` soit différent de 0 avant de lire `TTY_READ`. `NTTYS` est un `#define` défini dans le Makefile de compilation avec le nombre de terminaux du SoC (en utilisant l'option `-D` de gcc). (C9 S10)

```
struct tty_s {
    int write;           // tty's output
    int status;          // tty's status something to read if not null
    int read;            // tty's input
    int unused;          // unused
};
extern volatile struct tty_s __tty_regs_map[NTTYS];
// extern : parce que ce tableau n'est pas dans ce fichier
// volatile : parce que le contenu du tableau peut changer tout seul, gcc doit le lire à chaque fois
// cela implique que gcc ne peut pas faire d'optimisation avec les registres du MIPS
// (cf. note en fin de page)
```

- En principe, cela ne devrait pas poser de difficulté, mais cela nécessite d'avoir compris comment fonctionne le TTY et de savoir écrire une fonction en C. Pour aider, au cas où, on peut donner la description de ce qui est attendu: Tant que le registre `status` est à 0 alors attendre, puis lire le registre `read`. Notez que cela nécessite de savoir accéder aux champs d'une structure.

```
int getchar0(void)
{
    while (__tty_regs_map[0].status == 0);
    return __tty_regs_map[0].read;
}
```

5. Écrivez la fonction C `int puts0(char *s)` qui écrit tous les caractères de la chaîne `s` sur le terminal TTY0. La fonction doit rendre le nombre de caractères écrits. On suppose que les registres des TTYs sont définis comme dans la question précédente.

```
int puts0(char *s)
{
    char *p = s;
    while (*p != '\0') {
        __tty_regs_map[0].write = *p; // p est une variable locale que l'on peut modifier
        p = p + 1;                    // *p désigne le caractère pointé par p
    }                                 // on incrémente p, donc on pointe le caractère suivant
    return p-s;                       // finalement, on rend le nombre de caractères écrits
}
```

## 6. Usage de Make (optionnel)

Nous allons systématiquement utiliser des Makefiles pour la compilation du code, mais aussi pour lancer le simulateur du prototype **almo1**. Pour cette première séance, les Makefiles ne permettent pas de faire des recompilations partielles de fichiers. Les Makefiles sont utilisés pour agréger toutes les actions que nous voulons faire sur les fichiers, c'est-à-dire : compiler, exécuter avec ou sans trace, nettoyer le répertoire. Nous avons recopié partiellement le premier Makefile pour montrer sa forme et poser quelques questions, auxquels vous savez certainement répondre.

La syntaxe des **Makefile**s peut-être très complexe (c'est un vieux langage), ici nous ne verrons qu'une petite partie. Notez que le Makefile **voit** les variables du **shell** comme s'il les avait définies lui-même.

```
# Tools and parameters definitions
# -----
# -- options used by the prototype simulator
NTTY    ?= 2 #                default number of ttys

# -- tools
CC      = mipsel-unknown-elf-gcc #    cross-compiler MIPS
LD      = mipsel-unknown-elf-ld #    linker MIPS
OD      = mipsel-unknown-elf-objdump # desassembler MIPS
SX      = almo1.x #              prototype simulator (named almo1.x)

# -- All flags used for the gcc compiler
CFLAGS  = -c #                  stop after compilation, then produce .o
CFLAGS += -Wall -Werror #       near all C warnings that becoming errors
CFLAGS += -mips32r2 #           define of MIPS version
CFLAGS += -std=c99 #            define of syntax version of C
CFLAGS += -fno-common #         no use common sections for nostatic vars
CFLAGS += -fno-builtin #        no use builtin func of gcc (ie strlen)
CFLAGS += -fomit-frame-pointer # only use of stack pointer ($29)
CFLAGS += -G0 #                 do not use global data pointer ($28)
CFLAGS += -O3 #                 full optimisation mode of compiler
CFLAGS += -I. #                 dir. where include <file.h> are
CFLAGS += -DNTTYS=$(NTTY) #     number of ttys in the prototype

# Rules (here they are used such as simple shell scripts)
```

```
#
help:
@echo "\nUsage : make <compil|exec|clean> [NTTY=num]\n"
@echo "      compil : compiles all sources"
@echo "      exec   : executes the prototype"
@echo "      clean  : clean all compiled files\n"

compil:
$(CC) -o hcpua.o $(CFLAGS) hcpua.S
@$(OD) -D hcpua.o > hcpua.o.s
$(LD) -o kernel.x -T kernel.ld hcpua.o
@$(OD) -D kernel.x > kernel.x.s

exec: compil
$(SX) -KERNEL kernel.x -NTTYS $(NTTY)

clean:
-rm *.o* *.x* *~ *.log.* proc?_term? 2> /dev/null || true
```

5. Où est utilisé `CFLAGS` ? Que fait `-DNTTYS=$(NTTY)` et pourquoi est-ce utile ici ? (C9 annexe S8)

- Le compilateur C peut avoir beaucoup de paramètres. Définir une variable `CFLAGS` permet de les déclarer une fois au début et d'utiliser cette variable plusieurs fois dans le `Makefile`. Si on veut changer un argument, il suffit de le faire une seule fois. Ce genre de choses est nécessaire si on veut faire des `Makefile` faciles à lire et à faire évoluer.
- La variable `CFLAGS` est utilisée par `gcc`, il y a ici toutes les options indispensables pour compiler mais il en existe beaucoup, ce qui fait des tonnes de combinaisons d'options !
- `-DNTTYS=$(NTTY)` permet de définir et donner une valeur à une macro (ici définition `NTTYS` avec la valeur `$(NTTY)` comme le fait un `#define` dans un fichier C. Cette commande évite donc d'ouvrir les codes pour les changer.

6. Si on exécute `make` sans cible, que se passe-t-il ? (C9 annexe S6)

- On exécute la règle `help`. Mettre une règle `help` comme règle par défaut permet de documenter l'usage du `Makefile`, ce qui est plutôt une bonne pratique quand il y a beaucoup de cible et de paramètres. C'est d'autant plus vrai qu'on utilise les `Makefiles` comme des ensembles de `shell script`.
- C'est la première cible qui est choisie, donc ici c'est équivalent à `make help`. Cela affiche l'usage pour connaître les cibles disponibles.

Réponses non présentes dans les slides, mais utiles à savoir.

6. à quoi servent `@` et `-` au début de certaines commandes ?

- `@` permet de ne pas afficher la commande avant son exécution. On peut rendre ce comportement systématique en ajoutant la règle `.SILENT:` n'importe où dans le fichier.
- `-` permet de ne pas stopper l'exécution des commandes même si elles rendent une erreur, c'est-à-dire une valeur de sortie différente de 0.

7. Au début du fichier se trouve la déclaration des variables du `Makefile`, quelle est la différence entre `=`, `?=` et `+=` ?

- `=` fait une affectation simple
- `?=` fait une affectation de la variable si elle n'est pas déjà définie comme variable d'environnement du shell ou dans la ligne de commande de `make`, par exemple avec `FROM`
- `+=` concatène la valeur courante à la valeur actuelle, c'est une concaténation de chaîne de caractères.

#### Note sur le mot clé `volatile`

- Quand le programme doit aller chercher une donnée dans la mémoire puis faire plusieurs calculs dessus, le compilateur optimise en réservant un registre du processeur pour cette variable afin de ne pas être obligé d'aller lire la mémoire à chaque fois. Mais, il y a des cas où ce comportement n'est pas souhaitable (il est même interdit). C'est le cas pour les données qui se trouvent dans les registres de contrôleur de périphériques. Ces données peuvent être changées par le périphérique sans que le processeur le sache, de sorte qu'une valeur lue par le processeur à l'instant `t` n'est plus la même (dans le registre du périphérique) à l'instant `t+1`. Le compilateur ne doit pas optimiser, il doit aller chercher la donnée en mémoire à chaque fois que le programme le demande.
  - `volatile` permet de dire à `gcc` que la variable en mémoire peut changer à tout moment, elle est volatile. Ainsi quand le programme demande de lire une variable `volatile` le compilateur doit toujours aller la lire en mémoire. Il ne doit jamais chercher à optimiser en utilisant un registre afin de réduire le nombre de lectures mémoire (load). De même, quand le programme écrit dans une variable `volatile`, cela doit toujours provoquer une écriture dans la mémoire (store).
  - Ainsi, les registres de périphériques doivent toujours être impérativement lus ou écrits à chaque fois que le programme le demande, parce que c'est justement ces lectures et ces écritures qui commandent le périphérique.