



# SYSTEMES EMBARQUES 1

---

TP02 – INTRODUCTION A L'ASSEMBLEUR ARM

Joël Corpataux & Samuel Mertenat  
Classe T2f

Fribourg, le 7 octobre 2014  
13h – 16h35

## TABLE DES MATIERES

Buts et Objectifs du travail pratique .....	3
Analyse .....	3
Conception .....	4
Implémentation (codage de l'algorithme en assembleur) .....	5
Test et validation .....	5
Questions .....	6
Comment et suivant quel format les chaînes de caractères sont-elles codées dans la mémoire ? .....	6
Comment délimite-t-on une chaîne de caractères en mémoire ? .....	6
Quelles sont les adresses absolues des deux chaînes de caractères (source et dest) et de la variable size ? ..	6
Quel est l'outil qui permet de placer les chaînes à leur position mémoire absolue ? .....	6
Citer les modes d'adressage courants supportés par le contrôleur ARM9 ? .....	6
Comment le code assembleur que vous avez écrit est-il codé en binaire ? .....	7
Quelle est la taille du code assembleur que vous avez écrit ? Pourrait-il être réduit ? .....	7
Acquis .....	7
Problèmes .....	8
Conclusion .....	8
Annexe .....	8

# SYSTEMES EMBARQUES 1

## TP02 – INTRODUCTION A L'ASSEMBLEUR ARM

### BUTS ET OBJECTIFS DU TRAVAIL PRATIQUE

Ce deuxième travail pratique est l'occasion de mettre en pratique les connaissances apprises lors des dernières cours de théorie et de réaliser un convertisseur de caractères, de minuscules vers des majuscules. Il sera divisé en 3 phases bien distinctes ; la phase d'analyse, consacrée à la recherche d'informations quant au codage des caractères, des modes d'adressage et des différents algorithmes, la phase de conception, sur papier, du design détaillé du code et la dernière phase, qui consiste à tester et à valider le bon fonctionnement du travail effectué.

### ANALYSE

La tâche à réaliser dans ce TP consiste à réaliser un convertisseur de minuscules en majuscules. Pour ce faire, nous aurons besoin de comprendre comment fonctionne le codage des caractères dans la mémoire, ainsi que les différents modes d'adressage disponible sur la cible i.MX27.

Le codage des caractères se fait sur 8 bits / 1 octet ; il repose sur une table, la table « ASCII ». Les caractères minuscules sont compris entre 97 et 122 et les lettres majuscules, entre 65 et 90 ; le décalage est donc de 32 (-32 si minuscule → majuscule).

Regular ASCII Chart (character codes 0 - 127)									
000	(nul)	016 ► (dle)	032 sp	048 0	064 Ø	080 P	096 `	112 p	
001 ☉ (soh)	017 ◄ (dc1)	033 !	049 1	065 Å	081 Q	097 a	113 q		
002 Ⓢ (stx)	018 † (dc2)	034 "	050 2	066 B	082 R	098 b	114 r		
003 ▼ (etx)	019 !! (dc3)	035 #	051 3	067 C	083 S	099 c	115 s		
004 + (eot)	020 ¶ (dc4)	036 \$	052 4	068 D	084 T	100 d	116 t		
005 Ⓜ (enq)	021 \$ (nak)	037 %	053 5	069 E	085 U	101 e	117 u		
006 ♣ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v		
007 ▪ (bel)	023 ‡ (etb)	039 '	055 7	071 G	087 W	103 g	119 w		
008 ▣ (bs)	024 † (can)	040 (	056 8	072 H	088 X	104 h	120 x		
009 (tab)	025 ‡ (em)	041 )	057 9	073 I	089 Y	105 i	121 y		
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z		
011 ♂ (vt)	027 – (esc)	043 +	059 ;	075 K	091 [	107 k	123 {		
012 † (np)	028 L (fs)	044 ,	060 <	076 L	092 \	108 l	124		
013 (cr)	029 ↔ (gs)	045 -	061 =	077 M	093 ]	109 m	125 }		
014 Ⓢ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~		
015 Ⓢ (si)	031 ▼ (us)	047 /	063 ?	079 O	095 _	111 o	127 ò		

Figure 1: Aperçu de la table ASCII

Pour traiter la chaîne de caractères donnée en entrée et sélectionner caractère par caractère, on utilisera les instructions « ldrb » et « strb » (chargement / sauvegarde de 1 octet à la fois).

Chaque caractère sera chargé un à un, si sa valeur est inférieure à 97 (« a ») ou supérieure à 122 (« z »), il sera directement sauvegardé dans le registre de destination (r0) et la variable « size » incrémentée de 1. Si ce n'est pas le cas, le caractère est donc une minuscule ; elle sera transformée en majuscule en effectuant une addition de -32 (cf. figure 1).

## CONCEPTION

Le but du programme est de convertir chaque lettre minuscule en majuscule. Son déroulement se passera ainsi :

- Lecture caractère par caractère issu de la « source »
- Vérification de la valeur « max » (nombre de caractères maxima pouvant être traités)
- Vérification de ledit caractère :
  - [1-96] : rien à faire
  - [97-122] : en minuscule → conversion en majuscule par l'addition du décalage entre 'a' et 'A' (-32).
  - [123-...] : rien à faire
  - 0 : plus de caractère → fin de la source.
- Sauvegarde du caractère dans la « destination »

Le fonctionnement de l'algorithme est plus détaillé sur la figure 2. Pour vérifier le bon fonctionnement de notre raisonnement, nous avons, au préalable, développé une ébauche de code en Java, visible sur la figure 3, ci-dessous.

Le chargement et la sauvegarde des caractères se feront selon le principe de « Load & Store », aux moyens des instructions « ldrb » et « strb ». Le « b » signifiant « byte », pour le traitement de 8 bits à la fois (un caractère est codé sur 8 bits ; cf. table ASCII).

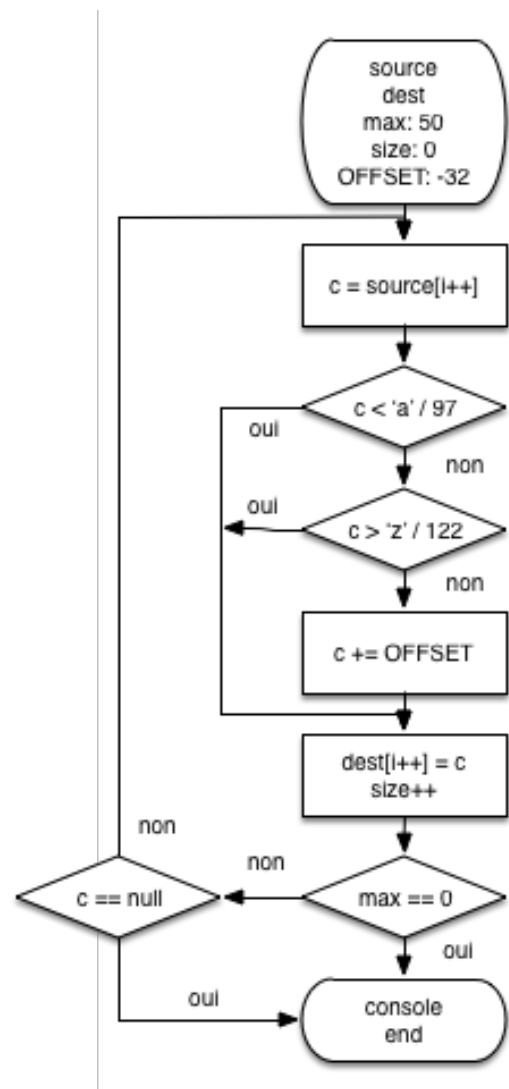


Figure 2: Fonctionnement de l'algorithme

```
package es02;

public class Main {

    public static void main(String[] args) {
        String str = "Halli haolo><)(7)%& HjugftRt%z `ab yz{\n";
        StringBuffer sb = new StringBuffer();
        char c;
        for (int i = 0; i < str.length(); i++) {
            c = str.charAt(i);
            if (c < 97)
                sb.append(c);
            else if (c >= 122)
                sb.append(c);
            else
                c -= 32;
            sb.append(c);
        }
        System.out.println(sb.toString());
    }
}
```

Figure 3: Vérification du principe en Java

Sortie sur la console :

```
HHALLI HAOL0>><<))((77]]%%&& HHJUGFTRRT%%zz ``AB Yzz{ {
```

## IMPLEMENTATION (CODAGE DE L'ALGORITHME EN ASSEMBLEUR)

Le code en assembleur de l'algorithme est disponible sur Git, dans le fichier « main.s ».

## TEST ET VALIDATION

Pour vérifier le bon fonctionnement de notre code, nous nous sommes aidés des outils mis à notre disposition sous la perspective « Debug ». Nous avons utilisé l'onglet « Registers » pour vérifier le bon chargement des caractères dans le registre « r4 » (valeur correspondant à une entrée dans la table ASCII), ainsi que du « Memory Browser », pour vérifier l'état de la mémoire.

(*)= Variables Breakpoints Registers	
Name	Value
General Registers	
r0	-1610605818
r1	-1610606334
r2	48
r3	-1610605824
r4	65
r5	2

Figure 4: Aperçu des registres lors de l'exécution du programme

## QUESTIONS

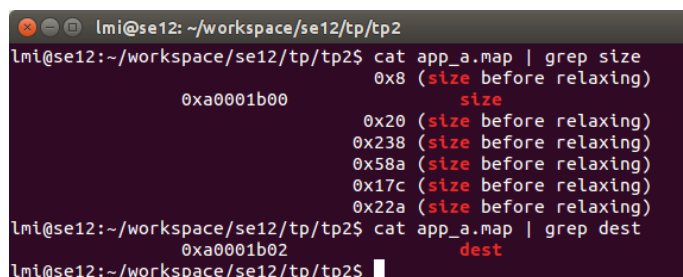
### COMMENT ET SUIVANT QUEL FORMAT LES CHAINES DE CARACTERES SONT-ELLES CODEES DANS LA MEMOIRE ?

Dans la mémoire, les caractères sont codés sur 8 bits, selon une table, appelée la table ASCII.

### COMMENT DELIMITE-T-ON UNE CHAINE DE CARACTERES EN MEMOIRE ?

Lors d'une instruction de déclaration, l'adresse du premier octet (un caractère équivaut à 8 bits / 1 octet) est enregistrée. Si c'est une chaîne (de caractères), la fin de celle-ci sera représentée par une valeur nulle (0x0, #0).

### QUELLES SONT LES ADRESSES ABSOLUES DES DEUX CHAINES DE CARACTERES (SOURCE ET DEST) ET DE LA VARIABLE SIZE ?



```
lmi@se12: ~/workspace/se12/tp/tp2
lmi@se12:~/workspace/se12/tp/tp2$ cat app_a.map | grep size
0x8 (size before relaxing)
0xa0001b00 size
0x20 (size before relaxing)
0x238 (size before relaxing)
0x58a (size before relaxing)
0x17c (size before relaxing)
0x22a (size before relaxing)
lmi@se12:~/workspace/se12/tp/tp2$ cat app_a.map | grep dest
0xa0001b02 dest
lmi@se12:~/workspace/se12/tp/tp2$
```

Figure 5: Adresses absolues de size et dest

Pour trouver les adresses mémoires des deux chaînes de caractères, ainsi que de la variable « size », on peut s'aider de la commande : « cat app\_a.map | grep <nom\_de\_la\_variable> », à taper à l'intérieur du dossier abritant le TP.

On trouve donc 0xa0001b02 pour « dest », 0xa0001b00 pour « size » et 0xa0001900 pour « source ».

### QUEL EST L'OUTIL QUI PERMET DE PLACER LES CHAINES A LEUR POSITION MEMOIRE ABSOLUE ?

Lors du processus d'implémentation du code, après l'assemblage de celui-ci, l'éditeur de liens attribue les espaces mémoires aux différentes parties du code (.data, .bss, .text, ...). Les adresses absolues restent donc les mêmes, tant que des instructions ne les affectent pas.

### CITER LES MODES D'ADRESSAGE COURANTS SUPPORTES PAR LE CONTROLEUR ARM9 ?

Nous pouvons citer deux modes d'adressages courants pour le contrôle ARM9 :

- « Load and Store » : ce mode consiste à d'abord charger les données dans des registres, d'y effectuer les opérations relatives, puis de stocker les données à nouveau dans la mémoire (« ldr » pour charger et « str » pour sauvegarder). Il existe plusieurs variantes : « Immediate offset », Register offset », « Pre-indexed » et « Post-indexed » (cf. p12, « Travaux pratiques TP2 »).

- « Data-processing operands » : ce mode permet d'assigner une valeur numérique constante (#, immediate shift) ou une valeur déjà contenue dans un registre (<Rd>, register shift) dans le registre voulu.

## COMMENT LE CODE ASSEMBLEUR QUI VOUS AVEZ ECRIT EST-IL CODE EN BINAIRE ?

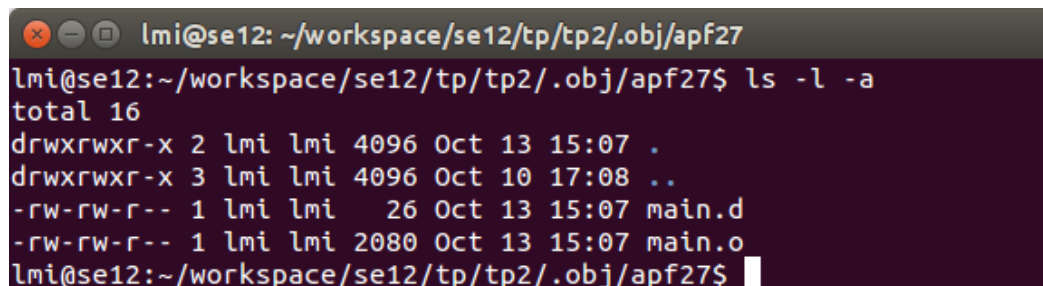
Comme on peut le voir sur le document présentant le TP2 (cf. page 12), les instructions sont codées sur 32 bits.

Exemple : `mov r5, #7`

**1110** (condition code, AI) **001** (Data processing : immediate) **1101** (Code d'opération : MOV) **0** (Statut : pas d'update) **0000** **0101** (Destination : r5) **0000** (Pas de rotation) **0000** **0111** (Valeur immédiate : 7) → 1110 0011 1010 000 0101 0000 0000 0111

## QUELLE EST LA TAILLE DU CODE ASSEMBLEUR QUE VOUS AVEZ ECRIT ? POURRAIT-IL ETRE REDUIT ?

Après compilation du code, le fichier « main.o » pèse 2080 octets. Cette information a pu être obtenue en allant dans « /workspace/se12/tp/tp2/.obj/apf275 », en utilisant la commande « `ls -la` » (-a : affichage des fichiers cachés ; -l : sous forme de liste avec informations).



```
lmi@se12: ~/workspace/se12/tp/tp2/.obj/apf27
lmi@se12:~/workspace/se12/tp/tp2/.obj/apf27$ ls -l -a
total 16
drwxrwxr-x 2 lmi lmi 4096 Oct 13 15:07 .
drwxrwxr-x 3 lmi lmi 4096 Oct 10 17:08 ..
-rw-rw-r-- 1 lmi lmi  26 Oct 13 15:07 main.d
-rw-rw-r-- 1 lmi lmi 2080 Oct 13 15:07 main.o
lmi@se12:~/workspace/se12/tp/tp2/.obj/apf27$
```

Figure 6: Aperçu des fichiers cachés issus du dossier TP2

La taille du fichier pourrait probablement être réduite, en utilisant des instructions « combinées » ou en supprimant commentaires et espaces inutiles.

## ACQUIS

Grâce à ce TP, nous avons pu mettre en pratique les notions théoriques vues précédemment en cours, comme l'adressage en mémoire, le principe de « Load and Store », les branchements, etc. De plus, nous avons vu comment fonctionne la table ASCII, ainsi que comment transformer une lettre minuscule en majuscule.

## PROBLEMES

Nous avons rencontré pas mal de difficultés à débiter le TP, mais après avoir relu la théorie et poser sur papier le problème, tout est devenu plus clair. D'autre part, nous avons eu quelques soucis à visionner le résultat issu de notre code, mais en se basant sur l'onglet « Registers » du menu « debug » et du « Memory Browser », tout est rentré dans l'ordre.

## CONCLUSION

Dans ce deuxième travail pratique, nous avons pu mettre un peu plus en pratique la notion de « Load and Store », ainsi que de comprendre comment sont stockés les caractères dans la mémoire et le fonctionnement de la table ASCII.

## ANNEXE

Le code source du programme a été déposé sur Git, avec un commit portant le nom de « main.s » (nom du fichier : main.s).