TP microcontrôleur 16 bits : MSP430

J.-M Friedt, 6 octobre 2008

Objectif de ce TP :
- présentation GNU/Linux et commandes unix
- faire clignoter des diodes, exemples en C et assembleur
- communication RS232, communication de données en ASCII
- lecture avec un baud rate et communication avec un autre
- communication SPI, accès à la carte SD

1 GNU/Linux

L'arborescence de GNU/Linux et comment s'y retrouver : dans l'ordre de priorité :
- `/` est la racine de l'arborescence. Le séparateur entre répertoire sera toujours le `/`.
- `/home` contient les répertoires des utilisateurs. Unix, et Linux en particulier, impose un confinement strict des fichiers de chaque utilisateur dans son répertoire. Seul l'administrateur (`root`) peut accéder aux fichiers du système en écriture : personne ne travaille jamais comme administrateur.
- `/usr` contient les programmes locaux à un ordinateur : applications, entêtes de compilation (`/usr/include`), sources du système (`/usr/src`)
- `/lib` contient les librairies du système d'exploitation
- `/proc` contient des pseudo fichiers fournissant des informations sur la configuration et l'état du matériel et du système d'exploitation. Par exemple, `cat /proc/cpuinfo`.
- `/etc` contient les fichiers de configuration du système ou la configuration par défaut des logiciels utilisés par les utilisateurs en l'absence de fichier de configuration locale (par exemple `cat /etc/profile`).
- `/bin` contient les outils du système accessibles aux utilisateurs et `/sbin` contient les outils du systèmes qu'*a priori* seul l'administrateur a à utiliser.

Les commandes de base :
- `ls` : list, afficher le contenu d'un répertoire
- `mv` : move, déplacer ou renommer un fichier. `mv source dest`
- `cd` : change directory, se déplacer dans l'arborescence. Pour remonter dans l'aroborescence : `cd ..`
- `rm` : remove, effacer un fichier
- `cp` : copy, copier un fichier. `cp source dest` Afin de copier un répertoire et son contenu, `cp -r repertoire destination`.
- `cat` : afficher le contenu d'un fichier. `cat fichier`
- `man`, manuel, est la commnde la plus importante sous Unix, qui donne la liste des options et le mode d'emploi de chaque commande.

Afin de remonter dans l'historique, utiliser SHIFT+PgUp. Toute commande est lancée en tâche de fond lorsqu'elle est terminée par `&`.

L'interface graphique est une surcouche qui ralentit le système d'exploitation et occupe des ressources, mais peut parfois faciliter la vie en proposant de travailler dans plusieurs fenêtres simultanément. Elle se lance au moyen de `startx`.

Parmi les outils mis à disposition, un éditeur de texte (`scite`), un clone opensource de matlab nommé `octave`, un outil pour tracer des courbes (`gnuplot`), affichage des fichiers au format PDF par `xpdf`, un gestionnaire de fichiers (`pcmanfm`).

La liste des modules noyaux (équivalent des drivers, chargés de réaliser l'interface logicielle entre le noyau Linux et l'espace utilisateur) s'obtient par `/sbin/lsmod`.

2 Premier pas avec gcc

Dans un éditeur de texte (`scite`), tapper le programme suivant :

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

int main() {printf("hello world %d\n",getpid());}
```

Nous n'utiliserons pas d'environnement intégré de développement (IDE) : un éditeur de texte sert à entrer son programme, et une fenêtre de commande (`xterm` ou Termial) servira à la compilation.

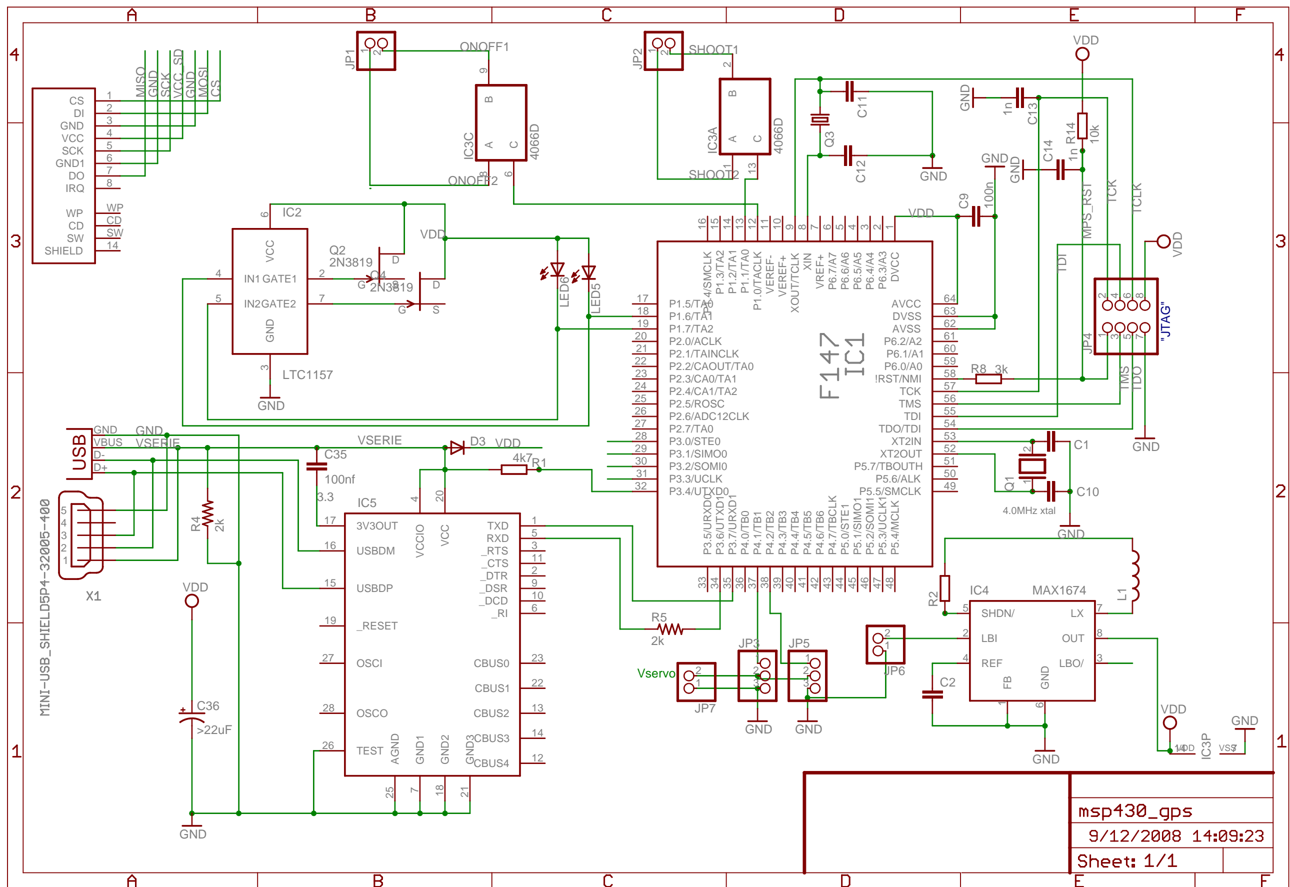
Une fois le programme tappé, sauver dans un fichier `exemple1.c` et compiler dans un terminal en entrant la ligne de commande : `gcc -o exemple1 exemple1.c` (compiler le programme C `exemple1.c` résultant en l'exécutable `exemple1`).

Exercice : exécuter le programme par `./exemple1`.

3 Premier pas sur MSP430

L'exemple précédent était une *compilation* de programme : l'architecture sur laquelle le programme sera exécuté est la même que celle sur laquelle le progamme est compilé (ici, Intel x86).

Nous allons désormais compiler des programmes à destination du microcontôleur : il s'agit de *cross-compilation*.



Le MSP430 [1] possède 16 registres dont 4 sont occupés pour des fonctions spécifiques (notamment R1=*stack pointer* : il reste donc 12 registres disponibles pour notre programme, R4-R15).

```
#include <signal.h>
#include <io.h>

#define         SEC     R13

.global main
main:
RESET:  MOV.W   #0x280,R1
        MOV.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop Watchdog Timer
        MOV      #0x0,&P1OUT
        MOV      #0x0,&P1SEL            ; p.9-4: P1_1 is TA, NOT GPIO
        MOV      #0xFF,&P1DIR
        MOV      #0,&P1IES              ; jmf
        MOV      #0,&P1IE               ; jmf

Mainloop:
        xor.b   #0xff,&P1OUT
        call    #delai
        jmp     Mainloop

delai:
        mov.w   #0x4fff,R7
attend: dec     R7
        jnz     attend
        ret
```

XOR	0	1
0	0	1
1	1	0

[1] focus.ti.com/lit/ds/symlink/msp430f149.pdf et http://focus.ti.com/lit/ug/slau049f/slau049f.pdf

Compilation du programme :
`msp430-gcc -Os -mmcu=msp430x149 -D_GNU_ASSEMBLER_  -o sortie.elf entree.S`

pour générer le listing correspondant :
`msp430-objdump -dSt led.elf > led.lst`

et la liste d'instructions au format hexadécimal Intel à partir du fichier ELF :
`msp430-objcopy -O ihex led.elf led.hex`

Rappel : `gcc -S` arrête la compilation à la génération de l'assembleur (fichier .s), `gcc -o` arrête la compilation à la génération des objets (avant le linker).

Chaque port d'entrée sortie nécessite de définir :
- la fonction de chaque broche avec le registre PxSEL : 0 pour du GPIO, 1 pour la fonction spécifique
- la direction du port dans PxDIR : 0 pour une entrée, 1 pour une sortie
- le niveau de la broche, 1 pour un niveau haut (3,3 V) et 0 pour le niveau bas (masse)

Exercices

1. où se trouve la pile dans cet exemple ?
2. combien de données pouvons nous placer au maximum sur la pile ?
3. proposer une alternative à l'opérateur XOR pour atteindre le même résultat.

Les ports du MSP430 sont capables d'absorber ou fournir la vingtaine de milliampères nécessaires à alimenter une LED.

Cet exemple exploite la fonction XOR pour périodiquement changer l'état du port et donc allumer ou éteindre la diode qui y est connectée.

Le même résultat s'obtient en C par :

```
#include  <msp430x14x.h>
#include <io.h>

main (void)
{volatile int i;
 P1OUT=0x00;
 P1SEL=0x00;
 P1DIR=0xFF;
 P1IES=0;
 P1IE=0;

 while (1)
  {
   P1OUT^=0xFF;
   for (i=0x3FFF;i>0;i--) {}
  }
}
```

Compilation du programme :
`msp430-gcc  -Os -mmcu=msp430x149 -o led led.c`

Exercices :

- visualiser le code assembleur généré
- comment est initialisé le pointeur de pile ? commenter ce choix.
- comparer l'efficacité du code C par rapport au code assembleur, à fonctionnalités égales, proposé plus haut.

4 Communiquer par RS232

Initialisation d'un port série : pour le port 0,

```
SetupUART0:     mov.b #SWRST,&UCTL0      ; cf p.13-4
                bis.b #CHAR,&UCTL0       ; 8-bit characters: 13-22
                mov.b #SSEL0,&UTCTL0     ; UCLK = ACLK @ 32768 Hz
                mov.b #0x03,&UBR00       ;
                mov.b #0x00,&UBR10       ; cf p.13-16: 9600 bauds +/- 15%
                mov.b #0x4A,&UMCTL0      ;
                bic.b #SWRST,&UCTL0
```

```
              bis.b #UTXE0+URXE0,&ME1 ; Enable USART0 TXD/RXD
              ret
```

ou pour le port 1 :

```
SetupUART1:   mov.b #SWRST,&UCTL1     ; cf p.13-4
              bis.b #CHAR,&UCTL1
              mov.b #SSEL0,&UTCTL1
              mov.b #0x03,&UBR01
              mov.b #0x00,&UBR11
              mov.b #0x4A,&UMCTL1
              bic.b #SWRST,&UCTL1
              bis.b #UTXE1+URXE1,&ME2
              ret
```

Une fois le port initialisé, la communication de données se fait en émission par :

```
rs_tx:        ;bit.b  #UTXIFG0,&IFG1  ; p.13-29: UTXIFG0 set if U0TXBUF empty
              ;jz     rs_tx
              ;mov.b  tmp,&TXBUF0     ;
              bit.b   #UTXIFG1,&IFG2
              jz      rs_tx
              mov.b   tmp,&TXBUF1     ;
              ret
```

ou en réception :

```
rs_rx:
        ; bit.b #URXIFG0,&IFG1  ; p.13-29: UTXIFG0 set if U0TXBUF empty
        bit.b   #URXIFG1,&IFG2  ; p.13-29: UTXIFG0 set if U0TXBUF empty
        jz     rsrx             ; pas de char recu
fin_rsrx:
        ; mov.b &RXBUF0,tmp
        mov.b   &RXBUF1,tmp
        ret
```

Côté PC, la communication par port série se fait au moyen de `minicom` : `CTRL A-O` pour définir le port (`/dev/ttyUSB0`) et le contrôle de flux (ni matériel, ni logiciel). `CTRL A-P` pour définir la vitesse (ici 9600 bauds, N81).

Exercices :

1. émettre en continu la valeur binaire correspondant au caractère 'A', 0x41 ou 65 en décimal.
2. envoyer une valeur hexadécimale (par exemple 0x55 et 0xAA) en ASCII
3. envoyer une valeur en décimal (par exemple 250) en ASCII
4. lire une caractère et le retranscrire
5. lire dans un baud rate (4800 N81) sur un port et écrire dans un autre (9600 N81) sur l'autre port
6. dans le programme en C faisant clignoter les diodes, ajouter une fonction pour visualiser le contenu de la pile et son évolution au cours de l'exécution du programme. Il s'agit d'une fonctionnalité de base des *debuggers*.

5 Lire une valeur analogique

Pour lancer une conversion analogique-numérique :

```
SetupADC12:   ; book C. Nagy p.90   Vtemp=0.00355(T_C)+0.986
              mov.w   #SHT0_6+SHT1_6+REFON+ADC12ON,&ADC12CTL0 ; V_REF=1.5 V
              mov.w   #SHP,&ADC12CTL1
              mov.b   #INCH_10+SREF_1,&ADC12MCTL0 ; p.17-11: single conversion
              bis.w   #ENC+ADC12SC,&ADC12CTL0         ; MUST be .w jmfriedt
adc:          bit.b   #1,&ADC12CTL1   ; wait while ADC busy==1
              jnz     adc
              mov.w   &ADC12MEM0,tmp  ; conversion result
              ret
```

Le canal 10 (`INCH_10`) est un cas particulier : il s'agit d'une sonde interne de température.

Exercice :
lire la température et la tranmettre sous forme hexadécimale (caractères ASCII) sur le port série.

La conversion s'obtient par (User Manual p.17-16) par $V_{Temp} = 0,00355 \times Temp_{degC} + 0,986$. Nous observons expérimentalement un décallage entre la température et la valeur du microcontrôleur qu'il semble nécessaire de calibrer pour chaque nouveau composant.

En C :

```
unsigned int IntDegC;

ADC12CTL0=SHT0_6+SHT1_6+REFON+ADC12ON; // V_REF=1.5 V
ADC12CTL1=SHP;
ADC12MCTL0=INCH_10+SREF_1;             // p.17-11: single conversion
for (j=0;j<16;j++) {
 ADC12CTL0 |= ENC+ADC12SC;
 do {} while ((ADC12CTL1&1)==1);
 IntDegC += (ADC12MEM0);
}
IntDegC/=16;
```

6 Interruption timer

Un gestionnaire d'interruption se définit par

```
interrupt(TIMERA0_VECTOR)               ;register interrupt vector
.global CCR0INT                         ;place a label afterwards so
CCR0INT:
                RETI                    ;
```

Ici l'interruption correspondant au timer A est activée chaque fois que le timer atteint une valeur prédéfinie :

```
setupTA:        MOV     #TASSEL0+TACLR, &TACTL  ; ACLK for Timer_A.
                BIS     #CCIE,&CCTL0            ; Enable CCR0 interrupt.
                MOV     #0x7FFF,&CCR0           ; load CCR0 with 32,767.
                BIS     #MC0,&TACTL             ; start TA in "up to CCR0" mode
```

Dans cet exemple, le timer A est initialisé à 0x7FFF=32767. Le timer étant cadencé par un quartz horloger à 2^{15} Hz, l'interruption est activée chaque seconde.

Une utilisation très utile du timer est la sortie périodique du mode veille dans lequel on place le microcontrôleur afin d'économiser l'énergie.

Exercices :

1. exploiter une interruption timer pour faire clignoter une diode à fréquence précisément déterminée depuis la fréquence du résonateur à quartz
2. écrire une fonction d'horloge temps-réelle capable d'afficher le temps en secondes et minutes
3. utiliser une représentation en BCD au lieu de binaire.

Afin d'effectuer une somme en BCD :

```
Clock:          SETC                    ; Set Carry bit.
                DADC.b  SEC             ; Increment seconds decimally
                CMP.b   #0x60,SEC       ; One minute elapsed?
                JLO     Clockend        ; No, return
                CLR.b   SEC             ; Yes, clear seconds
                DADC.b  MIN             ; Increment minutes decimally
                CMP.b   #0x60,MIN       ; Sixty minutes elapsed?
                JLO     Clockend        ; No, return
                CLR.b   MIN             ; yes, clear minutes
                DADC.b  HR              ; Increment Hours decimally
                CMP.b   #0x24,HR        ; 24 hours elapsed?
                JLO     Clockend        ; No, return
                CLR.b   HR              ; yes, clear hours
Clockend:       RET                     ;
```

7 Exemples en C

La programmation en C sur microcontrôleur se résume en grande partie à une syntaxe plus compacte mais une connaissance détaillée du fonctionnement du matériel et de la fonction de chaque bit de chaque registre reste nécessaire.

```
#include "hardware.h"
#include <signal.h>

void delay(unsigned int d) { int i; for (i = 0; i<d; i++) { nop(); nop(); } }

int test=0;

int main(void) {
    int sec=0,min=0,hr=0;

    WDTCTL = WDTCTL_INIT;           //Init watchdog timer
    P1OUT  = P1OUT_INIT;            //Init output data of port1
    P1SEL  = P1SEL_INIT;            //Select port or module -function on port1
    P1DIR  = P1DIR_INIT;                //Init port direction register of port1
    P1IES  = P1IES_INIT;                //init port interrupts
    P1IE   = P1IE_INIT;
/*
 while (1) {
            P1OUT ^= 0xff; delay(0x4fff);
            P1OUT ^= 0xff; delay(0x4fff);
 }
*/
    TACTL  = TASSEL0+TACLR;
    //CCTL0  |= CCIE; // 0x10
    asm("BIS #0x10,&0x0162");
    CCR0   = 0x7FFF;
    //TACTL  |= MC0;  // 0x10
    asm("BIS     #0x10,&0x0160");
    eint();

    while (1)
    {// asm("bis     #208,   r2\n");

     sec++;
     if (sec>59) {min++;sec=0;}
     if (min>59) {hr++;min=0;}
     if (hr>23) hr=0;
     P1OUT = 0xff;
     do {asm("nop");} while ((TACTL&0x01)==0x00);
     P1OUT = 0x00; delay(0x4fff);
     test=0;
    }

}

 interrupt(TIMERA0_VECTOR) isr() {test=1;}
```

Exercices :

1. faire clignoter une diode avec un délai (programme en C)
2. faire clignoter une diode avec une interruption timer (programme en C)
3. communiquer sur le port RS232 (programme en C)

8 Accès carte mémoire

Les microcontrôleurs 8 et 16 bits ne fournissent pas une puissance de calcul importante mais sont capable d'effectuer un certain nombre d'opérations simples telles qu'asservissements et acquisitions de données. Dans tous les cas, par soucis de contrôle *a posteriori* ou de dupliquer en local les données transmises en temps réel, les applications de ces microcontrôleurs sont étendues par la possibilité de stocker ces informations sur un support peu coûteux et consommant peu de courant en mode veille (Fig. 2). Nous nous proposons d'étudier le stockage d'informations sur cartes mémoires de type MutliMediaCard (MMC) ou Secure Digital (SD).

Afin de gagner du temps, nous ne détaillerons pas le protocole d'initialisation et de communication d'une carte SD : nous utiliserons une librairie mise à disposition par son auteur pour conserver et relire des données sur support de stockage de masse non volatile [2]. L'utilisation d'une librairie opensource ne prive cependant pas d'en lire le contenu en vue d'en comprendre le fonctionnement.

Les cartes SD fonctionnent en deux mode : le mode natif, rapide et supportant l'encryption des données, et le mode SPI, plus lent mais facilement accessible depuis n'importe quel microcontrôleur. Nous utiliserons ce second mode, dans lequel seules 4 broches sont utiles (fig. 1).

[2] http://www.true-random.com/homepage/projects/msp430_mmc/

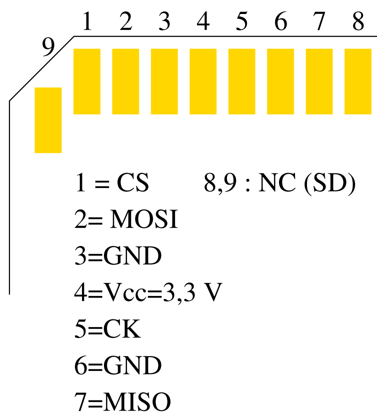


Fig. 1 – Assignation des broches d'une carte SD. Les broches 8 et 9 n'existent pas dans les cartes MMC, qui fonctionnent sinon de la même façon.

Après initialisation par `if (initMMC() == MMC_SUCCESS)`, deux commandes de base nous servent à écrire (`mmcWriteBlock(adresse)`) et lire (`mmcReadBlock(adresse, taille)`) des données sur la carte. Il s'agit d'écritures non-formatées : la carte n'est pas lisible depuis un PC qui s'attend à trouver un système de fichier. Toute transaction sur une carte SD se fait par blocs de 512 octets (tableau `mmc_buffer`). Les informations ne sont pas stockées en zone non-volatile tant que l'acquittement d'écriture n'est pas émis : lors des acquisitions lentes, une partie de la dernière séquence d'acquisitions pourra être perdue lors de la mise hors tension du circuit.

Afin de compiler notre programme `programme.c` avec les librairies d'accès à la carte SD :

```
msp430-gcc -mmcu=msp430x149 -o sd.elf test_jmf.c led.o mmc.o
```

Pour programmer et voir immédiatement le résultat :

```
msp430-jtag -e sd.elf && stty -F /dev/ttyUSB0 9600 && cat < /dev/ttyUSB0
```

Exercices :

- valider quelques fonctions de debuggage dont nous devons être certain du bon fonctionnement : allumage/extinction des diodes, transmission de données en RS232
- écrire dans un bloc de la carte SD une série de valeurs, réinitialiser le tableau de caractères et lire les valeurs stockées. Vérifier qu'il s'agit des valeurs écrites.
- écrire et relire sur plusieurs blocs mémoire. Attention : les adresses doivent *toujours* être multiples de 512.
- écrire des valeurs issues de la conversion analogique-numérique ou de la lecture d'un port série. Que se passe-t-il lors du passage d'un bloc de la carte SD au suivant ?

Pour aller plus loin : implémenter un système de fichier qui permette la relecture des informations stockées sur la carte depuis un PC (FAT16).

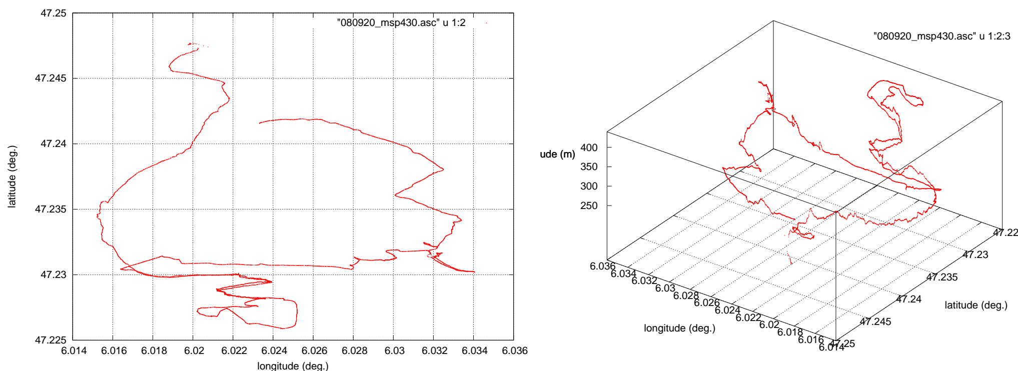


Fig. 2 – Exemple d'acquisition de trames GPS avec une fréquence de 1 Hz, et restitution du tracé en 2D (gauche) et 3D (droite) sous `gnuplot`.

Afficher des données

Deux outils sont mis à disposition pour afficher des courbes : `gnuplot` et `octave` [3].

`gnuplot` affiche des courbes lus dans des fichiers :

```
plot "fichier"
```

on peut préciser les colonnes à afficher, l'axe des abscisses et ordonnées... :

```
plot "fichier" using 2:3 with lines
```

pour utiliser la seconde colonne comme abscisse et la troisième colonne comme ordonnée, les points étant reliés par des lignes. `help plot` pour toutes les options.

`octave` est une version opensource de Matlab et en reprend toutes les syntaxes. Seule fonctionnalité nouvelle qui nous sera utile ici : `octave` peut lire des données au format hexadécimaal :

```
f=fopen("fichier","r");d=fscanf("%x");fclose(d);
```

La variable `d` contient alors les informations qui étaient stockées dans le fichier `fichier` au format hexadécimal. Si le fichier contenait `N` lignes, les données sont lues par colonne et les informations d'une colonne donnée d'obtiennent par `d(1 :N :length(d))`.

[3] Pour une description plus exaustive des fonctionnalités de `gnuplot` et `octave`, on pourra consulter `http://jmfriedt.free.fr/lm_octave.pdf`.