Examen M1 EEA 1er semestre électronique embarquée

Durée : 2 h, tous documents autorisés
6 décembre 2018

L'objectif de l'examen est de démontrer la capacité à exploiter une bibliothèque fournissant un certain nombre de fonctions, et ce dans un objectif de code portable séparant fonctions liées à un matériel spécifique et partie algorithmique de traitement du code.

Nous fournissons un squelette de programme qui ne contient que des fonctions indépendantes du matériel : ce programme se charge d'acquérir des données d'un convertisseur analogique numérique et les afficher à l'écran. Nous nous proposons dans cet exercice d'écrire les implémentations bas-niveau des fonctions appelées afin d'obtenir des codes complets fonctionnels. En vue de tester le code à la fois sur PC et sur microcontrôleur, l'implémentation des fonctions bas-niveau se fera dans des programmes spécifiques, dont les fichiers seront nommés `pc.c` (pour le PC) et `arm.c` (pour le microcontrôleur), auxquels on se liera lors de la compilation séparée. Pour ARM Cortex, nous fournissons les fonctions d'initialisation du port série et d'envoi d'un caractère (`usart_cm3.c`), initialisation du convertisseur analogique numérique et réception d'une mesure (`adc_cm3.c`) et initialisation et définition de l'état de GPIOs (`gpio_cm3.c`).

1. Démontrer votre capacité à mettre en œuvre les fonctions fournies dans les diverses bibliothèques contenues dans l'archive `/home/jmfriedt/examM1_2018.tar.gz` (3 points).

   Pour ce faire, dans la fonction principale du programme, créer une variable codée sur 16 bits nommée `res`. Cette variable doit être incrémentée de 0 à 10. À chaque incrément, la valeur contenue dans `res` doit être affichée sur le port série (UART1) du microcontrôleur. Pour faciliter la suite du développement, on placera les fonctions d'affichage dans un programme séparé `arm.c`, qui doit être lié à `main.c`. Le Makefile fourni permet de compiler le programme et de l'éxécuter sous `qemu` par `make exec`.

2. Démontrer votre capacité à mettre en œuvre les fonctions fournies dans les diverses bibliothèques pour acquérir et afficher les valeurs issues des mesures sur la voie 1 du convertisseur analogique-numérique du STM32 (3 points).
   - On modifiera le Makefile pour se lier aux fonctions de l'ADC
   - On démontrera le bon fonctionnement du programme en exécutant sous `qemu` par la commande proposée dans `make exec`.
   - En redirigeant la sortie du programme vers un fichier (`make exec > fichier` et en interrompant après quelques secondes son exécution (`CTRL-c`), tracer la courbe formée par la séquence de mesures. Quelle est la forme d'onde du signal simulé par la voie 1 du convertisseur analogique-numérique ?

   On se rappellera par exemple que pour interpréter des données affichées en hexadécimal, `gnuplot` requiert que les valeurs soient préfixées de "0x". Sous GNU/Octave, la lecture de valeurs hexadécimales s'obtient par
   `f=fopen('fichier');d=fscanf(f,"%x");fclose(f);`
   tandis que la lecture de valeurs décimales est immédiate dans les deux cas (`plot 'fichier'` dans gnuplot, `load fichier` dans GNU/Octave). Python peut aussi fournir les outils pour tracer la courbe résultante de l'acquisition.

3. Identifier la fréquence normalisée du signal (i.e. fréquence du signal par rapport à la fréquence d'échantillonnage) (2 point).

Ayant démontré notre capacité à acquérir les données depuis un émulateur de STM32, nous désirons simuler le comportement du système sur PC. Pour cela, nous implémenterons les fonctions simulant le convertisseur analogique-numérique unipolaire codant sa mesure sur 12 bits. Supposons que l'échantillonnage se fasse à 10 kHz et que le signal sinusoïdal acquis présente une fréquence de 500 Hz.

4. proposer les fonctions dans `pc.c`, dont les prototypes sont fournis dans `common.h`, permettant de compiler le programme `main.c` que vous venez d'écrire en question 1 ci-dessus sur PC (processeur x86, compilateur `gcc`) et l'exécuter sur PC. Seul l'affichage de `res` est demandé pour le moment (3 points).
5. démontrer l'acquisition de la courbe simulée en créant dans un fichier `adc_pc.c` les fonctions émulant le comportement de l'ADC. L'affichage et l'analyse de la courbe résultante permettra de se convaincre de l'exactitude de la fréquence du signal par rapport à la fréquence d'échantillonnage. Justifier. On se rappellera que l'exploitation des fonctions trigonométriques nécessite de se lier avec la bibliothèques mathématique par l'option `-lm` de `gcc` (3 points).
6. nous désirons induire un bruit sur le convertisseur analogique-numérique : pour ce faire, nous utiliserons la fonction `rand()` qui génère un entier aléatoire selon le prototype décrit dans la page du manuel `man 3 rand`. Sachant que tous les fichiers d'entête définissant toutes les constantes se trouvent dans `/usr/include`, quelle est la valeur maximale que peut fournir `rand()` ? (1 point)
7. ajouter du bruit blanc au signal sinusoïdal généré auparavant afin d'induire un rapport signal à bruit de 10. Justifier (2 points).

Questions de cours : (5 points)

8. quel attribut de définition d'une variable la place sur le tas ?
9. quelle est la conséquence sur la durée de vie de cette variable ?
10. quel attribut de définition d'une variable interdit au compilateur de faire une hypothèse sur sa valeur et donc d'optimiser le code qui lui est associé ?
11. donner deux cas d'utilisation de cet attribut.
12. quel est le temps de programmation de la fréquence de sortie d'un DDS AD9834 si son bus SPI est cadencé à 16 MHz ?

Correction :

Ce sujet d'examen a été l'occasion de corriger une erreur dans `qemu` pour STM32 qui ne gérait pas convenablement la variable de déclenchement de la conversion analogique numérique [1]

```c
#include "common.h"

int main(void)
{ short res;
  Usart1_Init();
  Led_Init();
  adc_setup();
  while (1) {
    res=read_adc(1);
    uart_putc('0');
    uart_putc('x');
    affshort(res);
    uart_puts("\r\n\0");
  }
  return 0;
}
```

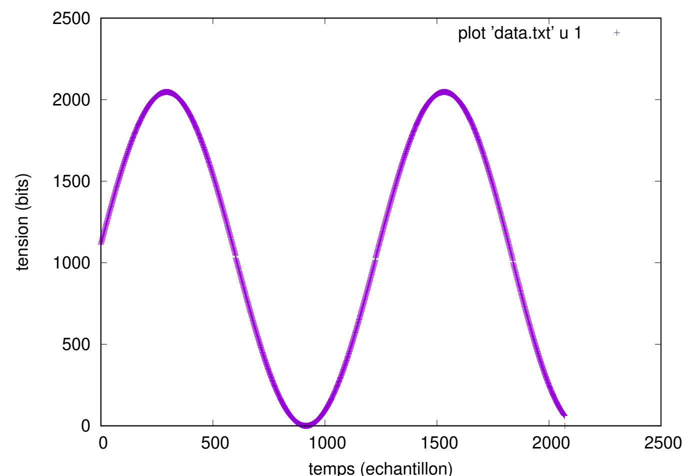
en exploitant les fonctions de `affichage.c` sur lequel nous nous lions lors de la compilation

```c
#include "common.h"

void affchar(char c)
{char b;
 b=((c&0xf0)>>4); if (b<10) uart_putc(b+'0'); else uart_putc(b+'A'-10);
 b=(c&0x0f);      if (b<10) uart_putc(b+'0'); else uart_putc(b+'A'-10);
}

void affshort(short s)
{affchar((s&0xff00)>>8);
 affchar(s&0xff);
}
```

qui utilise les fonctions d'initialisation, d'acquistion (`read_adc`) et d'affichage (`uart_putc`).



Ce programme résulte, suite à son exécution dans `qemu` et tracé dans `gnuplot`, la courbe ci-contre. Nous constatons qu'une période occupe environ 1200 échantillons soit une fréquence de signal de $f_s/1200$ avec f_s la fréquence d'échantillonnage. En considérant $f_s = 1$ (tel que considéré par Matlab par exemple), la fréquence normalisée est 1/1200.

La séparation de `affichage.c` du programme principal, et le lien lors de la compilation séparée avec des programmes implémentant les accès aux ressources bas niveau (convertisseur analogique-numérique, communication asynchrone) permet d'émuler ces fonctions sur PC.

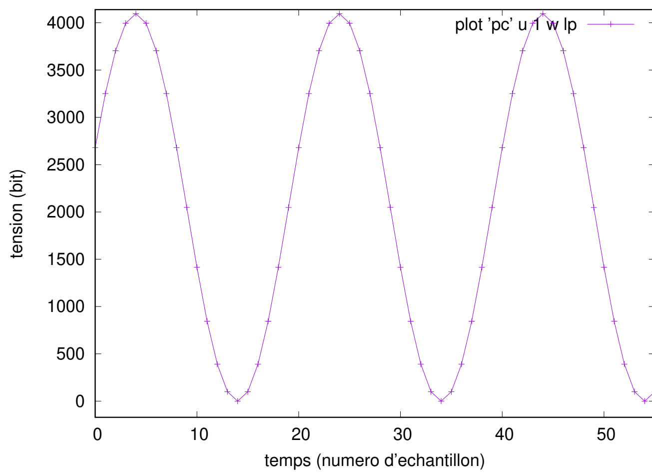
```c
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include "common.h"

static int temps;

void Usart1_Init(void) {printf("USART init\n");}
void Led_Init(void)    {printf("LED init\n");}
void Led_Hi1(void)     {printf("LED1 hi\n");}
void Led_Lo1(void)     {printf("LED1 lo\n");}
void Led_Hi2(void)     {printf("LED2 hi\n");}
void Led_Lo2(void)     {printf("LED2 lo\n");}
void uart_putc(char c) {printf("%c", c);}
void uart_puts(char *c){while(*c!=0) uart_putc(*(c++));}
void adc_setup(void)   {printf("ADC init\n");temps=0;}
unsigned short read_adc(unsigned char chan)
{temps++;
 return((short)(1024.*(1.+sin(2.*M_PI*temps*500./10000.))));
}
```

Le programme ci-contre implémente les diverses fonctions émulant le comportement du microcontrôleur.

1. https://github.com/beckus/qemu_stm32/commit/5a48797c9d8338f010f2aa4517596768fb6ad154



La simulation du convertisseur analogique numérique par le PC doit fournir 20 points/période (10000/500), tel qu'observé sur la figure ci-contre. Par ailleurs, un convertisseur analogique-numérique unipolaire ne fournit que des valeurs positives, et un convertisseur sur 12 bits fournit des mesures entre 0 et 2048-1.

La valeur maximale de la fonction `rand()` est données dans `stdlib.h` :

```
$ grep RAND /usr/include/* |& grep MAX
/usr/include/stdlib.h:#define   RAND_MAX        2147483647
```

Nous devons donc retrancher la valeur moyenne de ce bruit et le normaliser pour obtenir le rapport signal à bruit voulu.

Questions de cours :

- static
- durée d'exécution du programme (et non pas la durée d'exécition de la fonction déclarant la variable, qui par défaut la met sur la pile)
- volatile
- variable globale modifiée par interruption et accès à une ressource matérielle
- la fréquence est codée sur 32 bits (deux mots de 16 bits avec le code du registre à programmer et 14 bits des mots de poids fort et faible de la fréquence). 32 bits à 16 MHz nécessite 32/16 μs=2 μs