Sujet informatique embarquée – examen M1

J.-M Friedt, 25 mai 2016

Tous les documents sont autorisés, toutes les connexions à internet sont autorisées mais les communications par téléphone mobile sont **proscrites**. Chaque question sera notée sur 1 point. Il est probablement plus judicieux de persévérer sur un thème que se disperser sur les 3 axes d'investigations si le temps manque.

1 Lecture datasheet : le MSP430

Le MSP430 est un petit microcontrôleur, manipulant des données codées sur 16 bits, particulièrement orienté vers les applications faible consommation alimentées sur piles. Comme tout microcontrôleur, ce processeur accède à ses ressources en indiquant leur adresse et quelle configuration placer dans les divers registres définissant les fonctions de chaque périphérique.

En s'appuyant sur le manuel d'utilisateur fourni à `http://www.ti.com.cn/cn/lit/ug/slau049f/slau049f.pdf`, et en particulier son chapitre 13,

1. Rappeler la différence entre un protocole de communication série synchrone et asynchrone.
2. À quelle classe appartient le protocole RS232 ? justifier, et quelle est la conséquence sur la configuration des interlocuteurs ?
3. Quel(s) registre(s) configurer pour atteindre un débit de 57600 bauds, et quelle est l'erreur par rapport à la fréquence idéale, en supposant le microcontrôleur cadencé par un oscillateur à 4 MHz ? (2 points)
4. Exprimer la réponse à la question précédente en langage C (en particulier quelle est l'adresse de chaque registre et la taille de la donnée qui y est stockée).
5. Quelle est la durée de transmission d'un octet sur bus RS232 au format 8N1 à 57600 bauds ? À combien de cycles d'horloge cela correspond-t-il si le processeur est cadencé à 4 MHz ? commenter.
6. Est-ce que la capacité de communication est handicapée par cette l'erreur sur le débit de communication ? Justifier.

2 Protection des données

Nous avons vu que la gestion de données communes à plusieurs tâches nécessite un mécanisme de protection pour garantir la cohérence des informations. Un exemple classique est la production de données par une tâche chargée de l'acquisition d'informations, et la consommation de ces données par une seconde tâche. Cette division représente par exemple le partage du travail entre divers développeurs aux compétences complémentaires, avec une acquisition associée à des activités orientées bas niveau et une exploitation des données plutôt orientée traitement du signal.

7. Quelle structure de donnée/mécanisme permet de protéger une séquence d'opérations agissant sur une structure de données partagée par plusieurs tâches/threads pour garantir qu'un unique processus agit sur ces données à chaque instant ?
8. Dans un contexte de producteur/consommateur de données, combien de telles structures faudra-t-il pour gérer le tampon contenant les données ? Proposer un organigramme illustrant cette réponse dans le cas de deux tâches.
9. En supposant que la production d'informations est beaucoup plus lente que le traitement, mais que nous nous imposons de ne perdre aucune donnée, nous travaillerons avec deux tampons accédés alternativement. Quelle est la conséquence par rapport au cas précédent ? Justifier (2 points)

3 Du pointeur de fonction au *rootkit*

La sécurité d'un système Unix, et de Linux en particulier, tient en la séparation stricte des accès aux ressources entre les divers utilisateurs et l'administrateur. Nous avons vu que pour accéder au matériel, le noyau a tous les droits, encore supérieurs à ceux de l'administrateur qui ne peut qu'émettre des commandes agissant depuis l'espace utilisateur : par conséquent, un pilote modifiant les fonctionnalités du noyau donne à un utilisateur mal intentionné des pouvoir supérieurs à ceux de l'administrateur, pour peu qu'il arrive à insérer son module noyau. À titre d'exemple, les *rootkit* font une utilisation intensive des modifications d'appels systèmes transitant par le noyau pour cacher leur activité, à l'insu de l'administrateur. Nous nous proposons d'explorer une approche fonctionnelle sur noyaux 3.x et 4.x de Linux pour rediriger des fonctions sollicitées lors des appels systèmes.

Pour commencer, nous allons nous intéresser aux pointeurs de fonctions en espace utilisateur. Une utilisation courante des pointeurs de fonction consiste à définir une méthode commune à toutes les implémentations d'un protocole, par exemple `write()` ou `open()`, et de décliner les diverses implémentations de ces fonctions pour les diverses implémentations matérielles en les faisant pointer vers l'implémentation appropriée (voir par exemple l'implémentation du protocole Modbus à `http://www.freemodbus.org/`, avec la fonction `eMBInit()` de `mb.c` qui initialise les pointeurs de fonctions selon le protocole de communication utilisé). Un exemple plus épuré serait :

```
#include <stdio.h>
int toto(int x) {return (x+1);}
int tata(int x) {return (x+2);}
int main()
{ int (*ma_func)(int);
  ma_func = &toto;  // remplacer par &tata
  printf("%d\n",ma_func(1));
  return 0;
}
```

En assembleur, l'adressage indirect permet de modifier le *program counter* par une adresse fournie via un registre (au lieu d'une adresse absolue ou relative au point d'appel).

10. Compiler sur PC ce programme en faisant pointer `ma_func` tantôt vers `toto`, tantôt vers `tata`, et constater que le résultat varie : quelle est la sortie du programme selon la fonction appelée ?
11. L'assembleur x86 étant particulièrement pénible à lire, cross-compiler cette application pour processeur AVR 8 bits (par exemple Atmega32 – comment faire ?) et observer *dans le code assembleur* intermédiaire (quelle option de `gcc` permet de générer le code assembleur ?) quelle instruction est appelée pour sauter à `ma_func`. Quelle est la différence selon que `ma_func` pointe sur `toto` ou `tata` (2 points) ?
12. Expliquer comment l'instruction assembleur au cœur de cet appel de fonction se comporte (on se reportera à la liste des opcodes du processeur AVR disponible à `http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf` pour justifier du fonctionnement de l'instruction assembleur incriminée)
13. Qu'est-ce que l'argument Z de l'instruction mise en jeu ? Comparer avec la version par adressage immédiat de la même instruction (fournir sa mnémonique).

Le comportement d'un appel système est similaire à celui que nous avons vu lorsqu'un appel POSIX est transmis à un pilote pour faire appel aux entrées de la structure de données `file_operations` : un pointeur de fonction indique quelle fonction sera appelée pour répondre à la requête. L'ensemble des appels systèmes accessibles depuis l'espace utilisateur est décrit dans `/usr/include/i386-linux-gnu/bits/syscall.h`. Ainsi, contrairement au cas des modules noyau qui fournissent chacun leur structure de donnée pour leurs appels de fonctions, le noyau Linux propose une table qui inclut tous les appels de fonctions : cette table est décrite dans `/usr/include/asm-generic/unistd.h`

14. en observant la séquence des appels systèmes en préfixant une commande `mkdir` par l'instruction `strace`, quel est l'appel système spécifique à cette opération qui est sollicité lors de la création d'un répertoire sous GNU/Linux ? Quel est l'indice de cet appel système dans la table des appels systèmes ?

Une méthode classique [1] pour cacher un fichier ou un répertoire, voir des actions mal intentionnées, à l'administrateur système d'un ordinateur sous GNU/Linux, est d'insérer un module qui intercepte les appels systèmes et en modifie le comportement pour ne pas répondre aux sollicitiations correspondantes. À titre d'exemple, le module noyau disponible à `http://jmfriedt.free.fr/module_M1.c` intercepte un appel système et le détourne de sa fonction d'origine. Quelques subtilités tiennent en ce que

— l'emplacement de la table contenant les pointeurs de fonctions n'est plus exportée par le noyau vers les modules. Nous devons donc chercher dans la mémoire du noyau où se trouve cette table. Le noyau Linux est stocké, sur architecture x86, à la fin de l'espace d'adressage virtuel, entre l'adresse `PAGE_OFFSET` et la fin de l'espace d'adressage virtuel [2]. Une recherche exhaustive permet de trouver l'emplacement du pointeur vers une des fonctions correspondant à un appel système connu, par exemple dans notre cas `close` (fonction `aquire_sys_call_table`),

```
unsigned long int offset = PAGE_OFFSET; // debut du kernel en RAM
unsigned long **sct;
while (offset < ULLONG_MAX) {
   sct = (unsigned long **)offset;
   if (sct[__NR_close] == (unsigned long *) sys_close) return sct;
   offset += sizeof(void *);
}
```

— ayant identifié l'emplacement de la table des fonctions correspondant aux appels systèmes, nous redirigeons une de ces entrées vers nos propres fonctions. Cependant sur architecture x86, cette opération est *a priori* empêchée par une configuration de la page mémoire contenant la table des pointeurs vers les fonctions système en lecture seule. Il faut donc d'abord passer la page mémoire en écriture avant de modifier la table des pointeurs de fonctions puis repasser la page mémoire en lecture seule (opérations sur le registre CR0 [3]).

```
original_cr0 = read_cr0();      // autorise l'ecriture sur la table d'appels
write_cr0(original_cr0 & ~0x00010000);
ref_sys_mkdir = (void *)sys_call_table[__NR_mkdir];
sys_call_table[__NR_mkdir] = (unsigned long *)new_sys_mkdir;
write_cr0(original_cr0);
```

15. Compiler le pilote proposé plus haut sur PC (processeur x86) et le charger en mémoire sur le PC. Fournir les message affiché lors du chargement.
16. Constater l'effet sur l'appel système qui a été intercepté (quel est-il ? comment démontrer l'interception de cet appel ?)
17. modifier le programme (fonction `new_sys_mkdir()` pour que malgré l'interception de l'appel système, l'utilisateur (ou l'administrateur) ait tout de même l'impression que son système fonctionne normalement, et ne puisse pas se rendre compte de la supercherie (2 points).

Nous constatons donc dans cet exercice qu'une compréhension détaillée du fonctionnement d'un microprocesseur, en particulier au niveau du noyau qui interagit avec le processeur comme le ferait un programme en C sur microcontrôleur, avec toutes les autorisations de manipulation des registres et de la mémoire, permet d'en manipuler à volonté les fonctionnalités et en particulier de retirer tout espoir de sécurité du système.

1. `turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html` ou `gadgetweb.de/linux/40-how-to-hijacking-the-syscall-table-on`
2. `http://www.arm.linux.org.uk/developer/memory.txt`
3. `http://wiki.osdev.org/CPU_Registers_x86#CR0`

Réponses

Lecture datasheet : le MSP430

1. Un protocole synchrone partage l'horloge qui cadence les transactions entre les deux interlocuteurs. Un protocole asynchrone ne partage pas l'horloge est suppose que les deux interlocuteurs communiquent au même débit.
2. Seuls 3 fils servent à échanger les données sur RS232 : émission, réception et masse. Il n'y a pas de partage d'horloge : les deux interlocuteurs doivent avoir été programmé avec la même cadence de communication pour pouvoir communiquer.
3. page 13-12 de la datasheet : `baudrate=BRCLK/N` avec `N={UBR11,UBR01}`. En choisissant BRCLK=SMCLK (sortie du quartz haute fréquence, page 4-2 de la datasheet indiquant que SMCLK est sélectionnable comme XT2CLK avec XT2CLK le quartz haute fréquence), nous avons $N=4 \times 10^6/57600 = 69 = 0x45$ donc `UBR01=0x45;` et `UBR11=0x00;`. En prenant la valeur entière 69, nous avons en pratique 57970 bauds, soit une erreur de 0,6%. Il est éventuellement possible d'ajuster finement la position temporelle de chaque bit en jouant sur UMCTL1, mais cet aspect dépasse le cadre de la question posée.
4. l'emplacement (page 13-21 de la datasheet) des registres UBRx{0,1} est 0x7{4,5} (ou 0x7{C,D} pour le second port série s'il existe) donc `*(unsigned char*)0x75=0x00;*(unsigned char*)0x74=69;` puisque chaque registre contient une valeur sur 8 bits (page 13-25)
5. 8N1 donc 1 bit de start, 8 bits de données, 1 bit de stop et pas de bit de parité, donc 10/57600=174 μs. Ceci correspond à près de 700 cycles d'horloge sur un MSP430. Comme l'exécution de chaque instruction sur MSP430 nécessite plusieurs cycles d'horloge, nous n'avons pendant ce temps que la capacité à exécuter une centaine d'instructions, mais le temps de communication reste très long devant le temps de traitement des informations.
6. Chaque transaction est initialisée par la transition du bit de start. Toute erreur de débit inférieure à 10% n'handicape pas la capacité de communication puisque l'horloge de l'UART est resynchronisée au début de chaque transaction.

4 Protection des données

7. sémaphore et sa version binaire, le mutex
8. deux mutex sont nécessaires, un qui indique la disponibilité des données (à l'issue de l'acquisition) et un qui indique la fin des traitements sur ces données. Les deux tâches productrice et consommatrice sont donc, en supposant deux sémaphores S1 et S2 initialisés :
   producteur : boucle { acquisition ;incrémente S1 ;décrémente S2 ;}
   consommateur : boucle { décrémente S1 ;traitement ;incrémente S2 ;}
   Ainsi, le producteur débloque le consommateur en fin d'acquisition, et se bloque tant que le traitement est en cours, tandis que le consommateur attend de recevoir suffisamment de données pour son traitement avant de débloquer le producteur en fin de traitement des données,
9. Une pile (FIFO) avec une condition de remplissage partiel (half-full) ou double-tampon permet d'informer le consommateur de la disponibilité de données à traiter sans pour autant interrompre le flux d'acquisition. Dans ces conditions, la première moitié (ou le premier) tampon est rempli tandis que le consommateur attend d'être débloqué (abaisse le sémaphore). Lorsque le premier tampon est plein, le producteur élève le sémaphore, le consommateur traite ce premier tampon tandis que le second est en cours de remplissage. En fin de traitement du premier tampon, le consommateur abaisse ce même sémaphore pour être réveillé en fin de remplissage du second tampon. Dans ces conditions, un seul sémaphore (au lieu de deux) est nécessaire.

5 Du pointeur de fonction au *rootkit*

10. selon que `ma_func` pointe sur `toto` ou `tata` le résultat affiché est 2 ou 3.
11. `avr-gcc -S` permet de générer le code assembleur, dans lequel nous observons la présence de `icall`, un appel de fonction par adressage indirect.
12. Dans ce cas, le registre Z contient l'adresse de la fonction appelée (voir `http://www.atmel.com/webdoc/avrassembler/avrassembler.wb_ICALL.html`)
13. Z est la concaténation de R31 et R30. L'instruction qui permet de sauter à une fonction adressée directement est `call`, qui prend en argument une constante (génée lors de la compilation pour valoir l'adresse de l'emplacement mémoire où est stockée la séquence d'opcodes de la fonction)
14. `strace mkdir toto` nous indique que l'appel système `mkdir("toto", 0777)` est appelé : c'est cet appel système que nous redirigerons pour intercepter la création de répertoires par un utilisateur (ou l'administrateur)
15. au chargement, le pilote fournit l'adresse à laquelle est stocké le noyau : c0000000
16. une fois le pilote chargé, l'appel système `mkdir` est intercepté et nous retournons une valeur nulle sans avoir créé le répetoire : il n'est plus possible de créer un répertoire. Par ailleurs, le pilote nous informe, dans les messages du système (`dmesg`), de l'interception de la requête de l'appel système (`intercept`), du nom du répertoire qui aurait dû être créé, et ses permissions.

17. la capacité de création de répertoire est simplement restaurée en appelant la fonction originale de création du répertoire – que nous avions pris soin de mémoriser dans **`ref_sys_mkdir`**, lors de l'appel de notre fonction de remplacement :

```
long new_sys_mkdir(const char __user *pnam, int mode)
{long ret=ref_sys_mkdir(pnam, mode);
 printk(KERN_INFO "intercept: %s:%x\n", pnam,mode);
 return 0;
}
```