TP ports de communication synchrones

É. Carry, J.-M Friedt

24 janvier 2017

Questions sur ce TP :

1. Quelle séquence d'octets définit la séquence d'initialisation d'une carte SD ?
2. Quel niveau placer sur le signal de sélection (*chip select*) d'un périphérique SPI tel que la carte SD pour l'activer ?
3. Quel est l'ordre des bits transmis sur une liaison SPI ? sur une liaison I^2C ?
4. Identifier l'adresse I^2C que nous transmettrons pour communiquer avec l'horloge temps-réel PCF8563 : où trouver cette information ? Cette valeur est-elle cohérente avec la documentation technique ? Justifier.
5. Quelle séquence de transactions permet de lire une température sur un LM74 ?
6. un code source Arduino, tel que celui fourni par exemple à l'adresse `http://tronixstuff.com/2013/08/13/tutorial-arduino-and-pcf8563-real-time-clock-ic/`, pour accéder à ce périphérique utilise une autre adresse : laquelle ?
7. Justifier cette différence, en suivant les appels de fonctions et en identifiant l'origine de la différence.
8. Quelle spécificité sur l'impédance du bus I^2C le différencie du bus SPI : quel composant ajouter sur le signal de données dans le contexte de son utilisation bidirectionnelle ?

1 Généralités sur les ports de communication synchrones SPI et I^2C

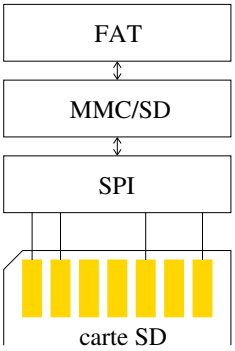
Les ports de communication synchrones permettent d'échanger des informations entre circuits physiquement distincts et fournissant des fonctionnalités complémentaires. Le mode le plus simple de communication est la liaison parallèle, dans laquelle N fils (un bus de N bits de largeur) portent les signaux pour échanger des informations sur N bits. Cependant, cette approche est gourmande en surface de circuit imprimé, en broches de circuits intégrés, et en cuivre de par la multiplicité des fils pour une liaison à longue distance. Par ailleurs, son débit est limité par le couplage capacitif entre pistes adjacentes. Ce mode de communication est donc souvent remplacé par une liaison série, dans laquelle les bits ne circulent pas en parallèle sur N fils mais sont séquencés dans le temps. Ces N intervalles de temps doivent donc être connus entre l'émetteur et le récepteur : dans le cas de l'UART il s'agissait d'un accord *a priori* (*baudrate*) entre les deux interlocuteurs dans le contexte d'une liaison asynchrone. Ici, nous nous intéresserons à deux protocoles largement déployés qui partagent l'horloge entre interlocuteurs (protocole synchrone) : SPI et I^2C. Le premier sépare les fils portant les signaux allant du maître à l'esclave et informe l'interlocuteur concerné par le message par un signal de *Chip Select*, le second ne fournit qu 'un fil bidirectionnel, faisant circuler adresse puis données.

2 Le connecteur UEXT

Les signaux qui nous intéressent sont tous routés vers le connecteur HE10 en bout de carte, nommé UEXT. Le schéma ci-dessous en fournit le brochage. Noter en particulier la possibilité de commuter l'alimentation du connecteur au travers d'un transistor, en vue de réduire la consommation globale du circuit en désactivant le périphérique qui y est connecté.
Analyser la figure 1, identifier le signal de commande de la grille du transistor, ainsi que les signaux disponibles pour la communication sur bus synchrone et asynchrone séries.

On notera que le FET est à canal P [1] et qu'il nécessite un potentiel négatif sur sa grille relativement à la source pour devenir passant.

3 Communication SPI



La maîtrise du bus SPI est fort utile, ne serait-ce que parce-que ce protocole permet de communiquer avec les cartes SD et donc d'étendre considérablement les capacités de stockage de nos circuits embarqués. Il est de façon générale utilisé par de nombreux capteurs – voir par exemple la note d'application AN1248 de Analog Devices à `http://www.analog.com/media/cn/technical-documentation/application-notes/AN-1248.pdf?doc=an-1250.pdf` – lorsque le débit de communication est modeste et la distance de communication faible (bus locaux à une carte ou un ensemble de cartes électroniques).
Le bus SPI sépare les transmissions du maître vers l'esclave (*Master Out Slave In* – MOSI) et esclave vers maître (*Master In Slave Out* – MISO). Par ailleurs, un signal de commande active le périphérique, CS#.
Rappeler la signification de la nomenclature # qui suit CS

1. `http://www.irf.com/product-info/datasheets/data/irlml6402.pdf`

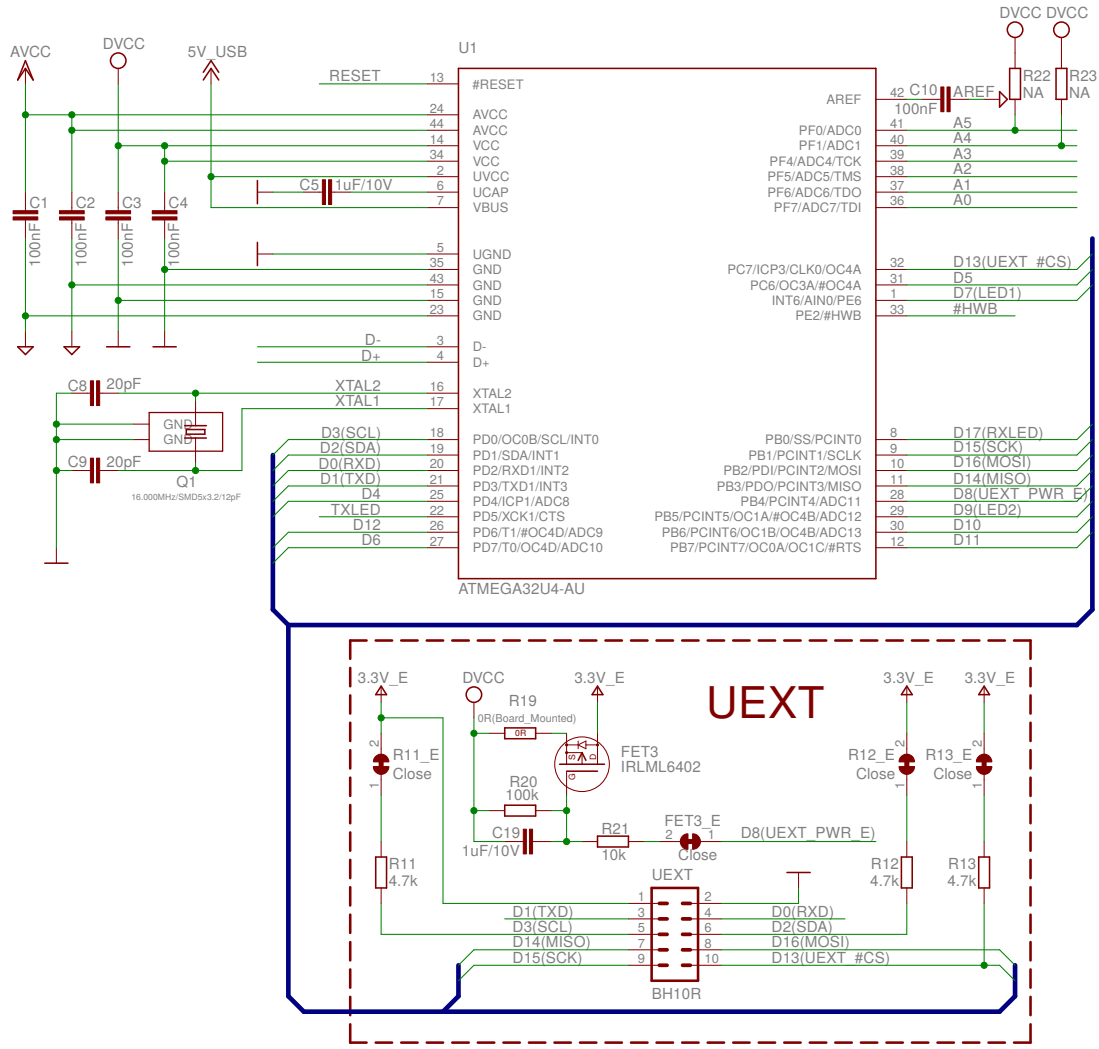


FIGURE 1 – Schéma du connecteur UEXT

Identifier, sur le schéma du connecteur UEXT, le GPIO qui sert de CS#.

Le choix d'implémenter CS# par un GPIO et non par le signal généré par l'USART de l'Atmega32U4 tient dans la souplesse de manuellement manipuler ce signal.
En l'implémentant sous forme de GPIO, nous choisissons quand nous abaissons et levons CS#, ce qui permet de transmettre plusieurs octets contigüs sans relever CS# et donc en informant l'interlocuteur de la continuité de la transaction.

4 Capteur de température

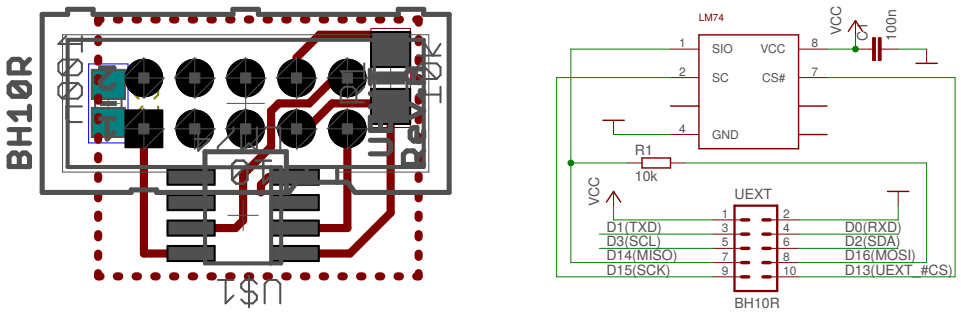


FIGURE 2 – Connexion du capteur de température LM74 au port UEXT de la carte Olimex et de son Atmega32U4.

Le LM74 est un capteur de température proposant une communication sur bus SPI (Fig. 2). Sa documentation technique nous informe de deux aspects de ses attentes concernant le protocole de communication :

1. l'état au repos de l'horloge,

2. le bit de donnée change d'état sur le front descendant de l'horloge et est stable sur le front montant. C'est donc sur cette seconde condition que le microcontrôleur échantillonnera l'état du bus.

Quel est le niveau au repos de l'horloge?

> When $\overline{CS}$ is high SI/O will be in TRI-STATE. Communication should be initiated by taking chip select ($\overline{CS}$) low. This should not be done when SC is changing from a low to high state. Once $\overline{CS}$ is low the serial I/O pin (SI/O) will transmit the first bit of data. The master can then read this bit with the rising edge of SC. The remainder of the data will be clocked out by the falling edge of SC. Once the 14 bits of data (one sign bit, twelve temperature bits and 1 high bit) are transmitted the SI/O line will go into TRI-STATE. $\overline{CS}$ can be taken high at any time during the transmit phase. If $\overline{CS}$ is brought low in the middle of a conversion the LM74 will complete the conversion and

Ces conditions sont déterminées par deux paramètres, CPHA et CPOL, que nous retrouverons sur toutes les implémentations de SPI. CPOL détermine l'état au repos de l'horloge : état haut si ce bit est à 1, état bas si ce bit est à 0. Le second aspect est déterminé par CPHA : si CPHA est à 1 alors la donnée s'établit sur le premier front d'horloge et s'est stabilisée sur le second front, lorsque le microcontrôleur échantillonne le bus. C'est cette condition qui nous intéresse, puisque le premier front d'horloge est le front descendant (repos à l'état haut) et le second front d'horloge est le front montant. Noter que le front sur lequel nous échantillons dépend de l'état au repos de l'horloge : le comportement serait inversé si nous laissions CPHA à 1 mais changions CPOL à 0.

Table 17-2. CPOL Functionality

CPOL	Leading Edge	Trailing Edge
0	Rising	Falling
1	Falling	Rising

- **Bit 2 – CPHA: Clock Phase**

The settings of the Clock Phase bit (CPHA) determine if data is sampled on the leading (first) or trailing (last) edge of SCK. Refer to Figure 17-3 on page 174 and Figure 17-4 on page 174 for an example. The CPOL functionality is summarized below:

Table 17-3. CPHA Functionality

CPHA	Leading Edge	Trailing Edge
0	Sample	Setup
1	Setup	Sample

FIGURE 3: Extrait de la *datasheet* de l'Atmega32U4 concernant les configurations du bus SPI.

Tracer le chronogramme d'une transaction en CPOL=0 et CPHA=1, se convaincre que dans ce cas l'échantillonnage se fait sur le front descendant de l'horloge.

Nous avons désormais toutes les informations pour configurer le bus SPI par son registre SPICR, qui contient par ailleurs le bit d'activation du bus et le facteur de division de l'horloge.

SPI est un bus qui manipule un bit définissant l'interlocuteur actif : une fois CS# abaissé, la transaction s'effectue en écrivant un octet dans SPIDR. MOSI et MISO étant séparés, ce même registre se remplit des bits transitant sur MISO alors que MOSI émet le mot à transmettre en même temps que les 8 coups d'horloge. En fin de transaction, CS# est remonté pour informer l'interlocuteur de la fin de la transaction.

Implémenter le protocole de communication avec le LM74, et afficher la séquence de bits lus. Est-elle cohérente avec la documentation technique? Si oui, convertir la séquence de bits en température (en degrés celsius, au millidegré près), et afficher son évolution (Fig. 5). La sortie du programme sera de la forme :

```
0000110011001100 0bcc 023.562
0000110011000100 0bc4 023.500
```

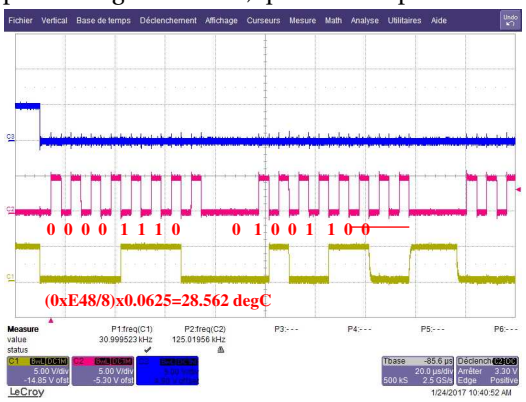


FIGURE 4: Chronogramme d'un échange sur SPI.

```
1  #include <avr/io.h>
2  #define F_CPU 16000000UL
3  #include <util/delay.h>
4  #include "VirtualSerial.h"
5
6  #define cs_lo PORTC &= ~(1 << PORTC7)     // CS#
7  #define cs_hi PORTC |= (1 << PORTC7)
8
9  extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
10 extern FILE USBSerialStream;
11
12 void init_SPI()
13 {DDRB |= ((1<<DDB0)|(1 << DDB2)| (1 << DDB1)); // MOSI (B2), SCK (B1) en sortie, MISO (B3) en entree
14  DDRB &= ~(1 << DDB3);    // ATTENTION : meme si CS manuel, DDB0 doit etre out pour ne pas bloquer SPI
15  DDRC |= (1 << DDC7);     // CS#                    // p.17.2.1 datasheet
16
17  SPCR = (0<<SPIE) | (1<<SPE)  | (0<<DORD) | (1<<MSTR) | (0 << CPOL) | (0<<CPHA) | (1<<SPR1) | (1<<SPR0);
18 //        Int Ena | SPI ENA   | 0=MSB 1st | Master    | CK idle hi  | sample trailing SCK |  f_OSC / 128
19  SPSR &= ~(1 << SPI2X); // No doubled clock frequency
20 }
21
22 char sd_raw_send_byte(char b)
23 {SPDR = b;                  // emet MOSI
24  while(!(SPSR & (1 << SPIF)));
25  SPSR &= ~(1 << SPIF);
```

```
 return SPDR;                // renvoie MISO
}

void binaire(char c,char *o)
{[... fonction qui remplit o avec les caracteres 0 ou 1 de la sequence binaire de c ...]}

int main()
{ char buffer[80]="C'est␣parti\r\n\0";
  unsigned char ch,cl;int k;
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();

  DDRB|=1<<PB4;                    // UEXT powered (FET)
  PORTB&=~1<<PB4;
  init_SPI();

  fputs(buffer, &USBSerialStream);
  while (1) {
    cs_lo;            // active la carte SD
    ch=sd_raw_send_byte(0x00);binaire(ch,buffer);
    cl=sd_raw_send_byte(0x00);binaire(cl,&buffer[8]);
    for (k=0;k<2;k++) {sd_raw_send_byte(0x00);}
    cs_hi;
    buffer[16]='\r';buffer[17]='\n';buffer[18]=0;
    fputs(buffer, &USBSerialStream);

    _delay_ms(100);
    CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
    CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
    USB_USBTask();
  }
  return(0);
}
```

Afin de se convaincre de l'importance de bien régler CPHA et CPOL, modifier volontairement CPOL pour lui donner une valeur erronée, et constater que les messages reçus ne sont plus stables. Expliquer.

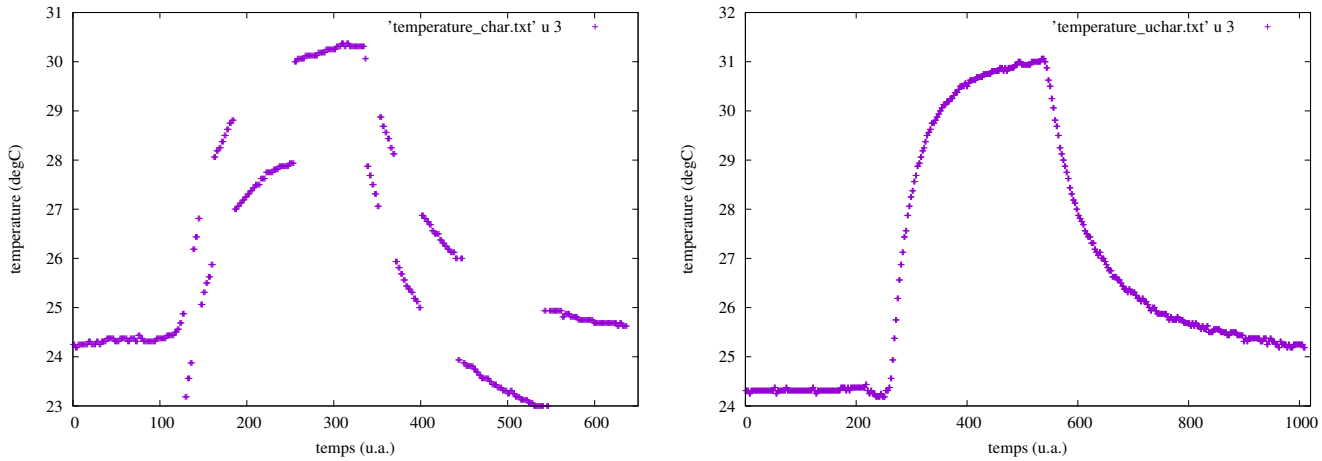


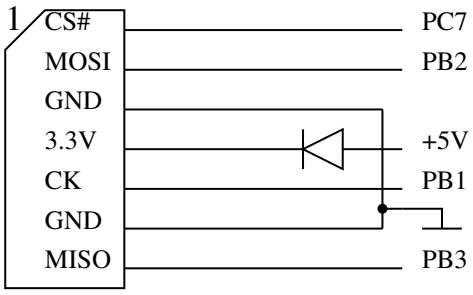
FIGURE 5 – Gauche : conversion de la séquence de bits en température en degrés Celsius. La concaténation de l'octet de poids fort avec l'octet de poids faible ne se passe pas correctement, lors de la multiplication par 256, si l'octet de poids fort est indiqué comme signé (comportement par défaut de `char`). Droite : le problème est résolu en précisant que les deux variables récupérant l'octet de poids fort et de poids faible décrivent des données non-signées. Dans ces deux exemples, le capteur passe de température ambiante à la température au contact de la paume de la main, puis retour à l'ambiante.

5 Carte SD

Afin de compléter notre connaissance du protocole SPI et appréhender le protocole de communication des cartes SD (en mode SPI), nous allons amorcer la première phase d'initialisation d'une carte SD au travers du code ci-dessous. On y

remarquera en particulier, en delà de l'initialisation du périphérique, la fonction `sd_raw_send_byte()` qui sert à la fois à émettre un octet (`SPDR=b;`) et, une fois la transaction achevée et 8 coups d'horloges transmis à l'esclave, de lire le résultat acquis sur la ligne MISO dans le registre à décallage par `return SPDR;`.

La connexion de la carte SD dans son mode de communication le plus simple (mais le moins performant – SPI) ne nécessite que 4 signaux en plus de la masse et de l'alimentation. Nous avons choisi la séquence de connexion proposée sur le schéma de droite, le seul degré de liberté portant sur CS# puisque nous exploiterons l'implémentation matérielle de SPI et donc les broches chargées des liaisons MISO, MOSI et CK sont imposées.



Quelle est l'utilité de la diode électroluminscente (LED) sur l'alimentation ? Quelle est la chute de tension liée à une LED ?

L'initialisation de la carte SD en mode SPI [2] s'obtient par la séquence suivante [1] :

1. Une série de coups d'horloge avec la ligne de données à l'état haut est fournie pour réveiller la carte : nous écrivons pour cela 10 fois la valeur 0xff sur le bus SPI avec CS# au niveau **haut**,
2. après avoir **abaissé** CS#, la commande CMD0 est transmise pour réinitialiser la carte SD. La commande CMD0 est formée d'un entête 0x40 auquel on ajoute le numéro de commande, puis de 4 arguments (nuls pour CMD0), et finalement un checksum, nécessaire pour le mode natif de la carte SD dans lequel nous nous trouvons actuellement, mais qui est précalculé à 0x95,
3. finalement, la commande CMD1 passe la carte SD de son mode natif au mode SPI, dans lequel le checksum n'est plus nécessaire. Nous conserverons arbitrairement 0x95 pour simplifier le code.
4. La carte SD répond 0xff tant que la commande n'est pas acquittée, et renvoie soit 0x00, soit 0x01 pour acquitter d'une commande. Ainsi, tant que la carte SD répond 0xff, nous lui redemandons sont statut, éventuellement jusqu'à atteindre un délai maximal (*timeout*). La réponse à CMD0 doit être 0x01, indiquant que la carte est en mode d'attente (*idle*). Pour les autres commandes, la réponse doit être 0x00, indiquant l'acquittement de l'ordre.

```
1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3  #define F_CPU 16000000UL
4  #include <util/delay.h>
5  #include "VirtualSerial.h"
6
7  #include <string.h>
8  #include <stdio.h>
9
10 #define cs_lo PORTC &= ~(1 << PORTC7)     // CS#
11 #define cs_hi PORTC |= (1 << PORTC7)
12 #define CMD_GO_IDLE_STATE 0x00
13 #define R1_IDLE_STATE 0
14
15 extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
16 extern FILE USBSerialStream;
17
18 void init_SPI()
19 {DDRB |= ((1<<DDB0)|(1 << DDB2)| (1 << DDB1)); // MOSI (B2), SCK (B1) en sortie, MISO (B3) en entree
20  DDRB &= ~(1 << DDB3);    // ATTENTION : meme si CS manuel, DDB0 doit etre out pour ne pas bloquer SPI
21  DDRC |= (1 << DDC7);              // CS#                    // p.17.2.1 datasheet
22
23  SPCR = (0<<SPIE) | (1<<SPE)  | (0<<DORD) | (1<<MSTR) | (0 << CPOL) | (0<<CPHA) | (1<<SPR1) | (1<<SPR0);
24 //        Int Ena | SPI ENA   | MSB 1st   | Master    | CK idle lo  | sample rising SCK |  f_OSC / 128
25  SPSR &= ~(1 << SPI2X); // No doubled clock frequency
26 }
27
28 char sd_raw_send_byte(char b)
29 {SPDR = b;                  // emet MOSI
30  while(!(SPSR & (1 << SPIF)));
31  SPSR &= ~(1 << SPIF);
32  return SPDR;               // renvoie MISO
33 }
34
35 unsigned char sd_raw_send_command(unsigned char command, unsigned long arg)
36 {unsigned char response,i;
37
```

2. http://elm-chan.org/docs/mmc/mmc_e.html

```
 fputs("\r\nenvoi␣cmd␣\0", &USBSerialStream);
 sd_raw_send_byte(0x40 | command);
 sd_raw_send_byte((arg >> 24) & 0xff);
 sd_raw_send_byte((arg >> 16) & 0xff);
 sd_raw_send_byte((arg >> 8) & 0xff);
 sd_raw_send_byte((arg >> 0) & 0xff);
 sd_raw_send_byte(0x95);  // CRC de la commande 0 -- CRC n'est plus utile apres

 for(i = 0; i < 10; ++i)
 { response = sd_raw_send_byte(0xff);
   fputs("ff\0", &USBSerialStream);
   if(response != 0xff) {fputs("\r\nfini\0", &USBSerialStream);break;}
 }
 return response;
}

int sd_raw_init()
{ int i;
  unsigned char response;
  for (i = 0; i < 10; ++i) sd_raw_send_byte(0xff); // 80 coups d'horloge

  cs_lo;           // active la carte SD
  i=0;
  do {i++;  // response est initialise' donc do au lieu de while
      response = sd_raw_send_command(CMD_GO_IDLE_STATE, 0);
     } while ((i<0x1ff) && (response!=(1 << R1_IDLE_STATE)));
  if (i == 0x1ff) { cs_hi; fputs("\r\nechec\0", &USBSerialStream);return 0; } // timeout
  return(1);
}

int main()
{ char buffer[80]="C'est␣parti\r\n\0";
  char c='\0';
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();

  do {
  if (CDC_Device_BytesReceived(&VirtualSerial_CDC_Interface))
     c=CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
  } while (c!='g');

  // sei();
  DDRB|=1<<PB4;                    // UEXT powered (FET)
  PORTB&=~1<<PB4;
  init_SPI();
  fputs(buffer, &USBSerialStream);
  sd_raw_init();
  while (1) {
    CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
    CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
    USB_USBTask();
  }
  return(0);
}
```

L'exécution de ce code, qui se contente de placer la carte SD dans le mode SPI par l'envoi de la commande 0, doit se traduire par

```
C'est parti

envoi cmd ff ff ff ff ff ff ff ff ff ff
envoi cmd ff ff
fini
```

qui indique que la carte SD s'est bien initialisée à la seconde tentative d'envoi de CMD0 (commande d'initialisation et de passage en mode SPI). Noter que le CRC de cette commande est tabulé, par exemple page 5-1 de `http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Components/General/SDSpec.pdf`, et n'a pas besoin d'être calculé explicitement. Le nombre

- **Bit 7 – SPIF: SPI Interrupt Flag**

When a serial transfer is complete, the SPIF Flag is set. An interrupt is generated if SPIE in SPCR is set and global interrupts are enabled. If $\overline{SS}$ is an input and is driven low when the SPI is in Master mode, this will also set the SPIF Flag. SPIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, the SPIF bit is cleared by first reading the SPI Status Register with SPIF set, then accessing the SPI Data Register (SPDR).

FIGURE 6: Signification du bit SPIF dans SPSR.

de `ff` varie puisqu'ils correspondent à l'absence de réponse pendant que la carte se configure. Nous cessons ici la démonstration de la communication pour ensuite faire confiance à une bibliothèque : en effet, les opérations suivantes deviennent dépendantes de la nature de la carte (MMC de faible capacité ou SD de forte capacité) et plus complexes à implémenter.

5.1 Lecture et écriture sur carte SD

Un exemple de bibliothèque en charge de la communication bas niveau mais aussi du protocole associé au stockage au format FAT16 détaillé est fourni à `http://elasticsheep.com/2010/01/reading-an-sd-card-with-an-atmega168`. Quelques points de détail concernant le passage à l'Atmega 32U4, en particulier la sélection des broches de communication et l'adaptation du `Makefile`, sont décrits à
`http://elasticsheep.com/2010/01/reading-an-sd-card-part-2-teensy-2-0`.

Télécharger `http://www.roland-riegel.de/sd-reader/sd-reader_source_20090330.zip` **et** `http://elasticsheep.com/wp-content/uploads/2010/01/sd-reader_source_20090330-teensy.patch.txt` **puis, afin d'appliquer le correctif (*patch*), se placer dans le répertoire issu de la décompression de l'archive .zip (commande** `unzip`**) et appliquer** `patch -p1 < emplacement_du_patch/sd*.txt`**.**

5.2 Mode polling

L'exemple proposé sur la page web ci-dessus est très instructif car il sépare proprement les aspects bas-niveau (interaction avec la matériel), la couche intermédiaire (protocole de communication sur bus SPI entre le microprocesseur et la carte SD) et la couche de haut niveau (protocole FAT16).

Ainsi, dans le fichier `sd_raw.c`, nous trouvons l'appel aux registres permettant d'effectuer, au niveau du matériel (registre du microcontrôleur), la transaction sur bus synchrone : nous y retrouvons dans `sd_raw_send_byte(uint8_t b)` et `uint8_t sd_raw_rec_byte()` deux fonctions de communication opérant de la même façon qu'observé auparavant.

Afin de tester la capacité à interagir avec la carte SD, nous allons aveuglément faire confiance en l'implémentation des couches hautes du protocole de communication, et ne nous intéresser qu'à la partie bas niveau concernant l'interaction du matériel avec le bus de communication UEXT.

Les liaisons sur le bus SPI sont standardisées entre les divers microcontrôleurs de la gamme Atmel 8-bits, et nous n'avons besoin que de modifier la configuration du signal de sélection de la carte : dans `sd_raw_config.h`,

```
#define configure_pin_ss() DDRC |= (1 << DDC7)    // PC7
#define select_card() PORTC &= ~(1 << PORTC7)     // CS#
#define unselect_card() PORTC |= (1 << PORTC7)
```

indiquent que le signal de sélection de périphérique se trouve connecté à PC7. La seule autre petite modification nécessaire consiste à remplacer la liaison USART par la liaison SPI en redéfinissant dans `uart.c` les fonctions

```
void uart_init()
{char c;
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();

  do {  // attend que l'utilisateur appuie sur 'g' pour commencer
    if (CDC_Device_BytesReceived(&VirtualSerial_CDC_Interface))
       c=CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
    } while (c!='g');
}

void uart_putc(uint8_t c) {fprintf(&USBSerialStream,"%c",c);}

uint8_t uart_getc()
{if (CDC_Device_BytesReceived(&VirtualSerial_CDC_Interface))
    return(CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface));
 else return(-1);
}
```

pour faire appel aux fonctions de la bibliothèque `LUFA` au lieu de la liaison sur port de communication asynchrone.
Prendre soin de modifier *tous* les éléments nécessaires à l'utilisation de `LUFA`**.**

Une carte SD travaille par blocs de 512 octets. Afin de tester la capacité de communiquer avec la carte SD, nous lirons le premier bloc, l'écraserons, le relirons, et reproduirons l'opération avec le second bloc (d'adresse 512, puisque les adresses sont indexées sur les octets individuels).

Ayant compris le fonctionnement bas niveau d'une carte SD, en particulier par son initialisation, nous allons désormais exploiter les fonctions de la bibliothèque téléchargée pour envoyer les commandes [3] – dont un extrait est reproduit ci-dessous – plus complexes permettant de stocker des informations. Le principe reste le même : une commande informe la carte SD de l'opération à effectuer, lecture ou écriture, suivi de l'adresse du bloc comme argument. Dans le cas d'une lecture ou écriture d'un bloc de données, les 512 octets qui suivent seront stockés par le support de masse non volatil avant qu'il renvoie deux octets de CRC.

Supported Commands	Abbreviation	SDMEM System	SDIO System	Comments
CMD0	GO_IDLE_STATE	Mandatory	Mandatory	Used to change from SD to SPI mode
CMD2	ALL_SEND_CID	Mandatory		CID not supported by SDIO
CMD3	SEND_RELATIVE_ADDR	Mandatory	Mandatory	
CMD4	SET_DSR	Optional		DSR not supported by SDIO
CMD5	IO_SEND_OP_COND		Mandatory	
CMD7	SELECT/DESELECT_CARD	Mandatory	Mandatory	
CMD9	SEND_CSD	Mandatory		CSD not supported by SDIO
CMD10	SEND_CID	Mandatory		CID not supported by SDIO
CMD12	STOP_TRANSMISSION	Mandatory		
CMD13	SEND_STATUS	Mandatory		Card Status includes only SDMEM information
CMD15	GO_INACTIVE_STATE	Mandatory	Mandatory	
CMD16	SET_BLOCKLEN	Mandatory		
CMD17	READ_SINGLE_BLOCK	Mandatory		
CMD18	READ_MULTIPLE_BLOCK	Mandatory		
CMD24	WRITE_BLOCK	Mandatory		
CMD25	WRITE_MULTIPLE_BLOCK	Mandatory		

Le code d'exemple peut servir d'inspiration, sans être un guide à suivre rigoureusement, pour stocker des informations connues sur certains blocs et ensuite relire le contenu pour valider que les informations ont bien été enregistrées sur la carte.

On notera que quelques modifications ont dû être apportées par rapport au code original pour s'adapter à l'Atmega32U4, pour s'adapter aux broches utilisées ainsi que pour remplacer la communication sur port RS232 par la liaison sur bus USB fournie par `LUFA`.

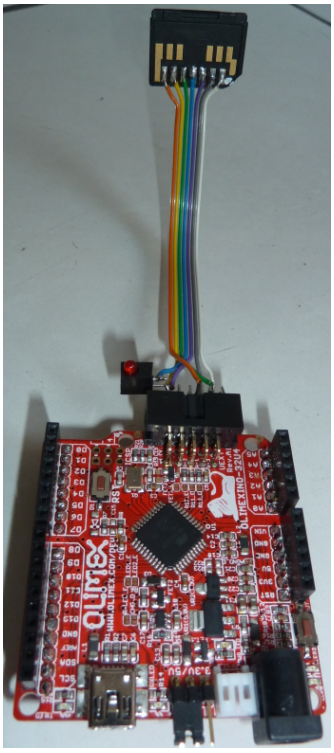
3. La liste des commandes comprises par une carte SD est disponible dans la documentation de `https://www.sdcard.org/downloads/pls/simplified_specs/archive/partE1_100.pdf` par exemple

```
// Atmega32U4
#if (BOARD == Olimex)
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#endif
// end Atmega32U4

#include "VirtualSerial.h"
extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;

int main()
{
#if (BOARD == Olimex)               // Atmega32U4
  uint8_t buffer[512]="Go\r\n\0";int k;
  DDRB|=1<<PB4;PORTB&=~1<<PB4;      // UEXT powered (FET)
  sei();
#endif // end of Atmega32U4
  set_sleep_mode(SLEEP_MODE_IDLE); // ordinary idle mode
  uart_init();                      // setup uart
  fputs(buffer, &USBSerialStream);
  if(!sd_raw_init())                // setup sd card slot
     {
#if DEBUG
      uart_puts_p(PSTR("MMC/SD initialization failed\n"));
#endif
      continue;
     } else
       uart_puts_p(PSTR("MMC/SD initialization success\n"));

/// jmf test
sd_raw_read(0x00,buffer,512);       // read first cluster
for (k=0;k<512;k++)
 {uart_putc(((buffer[k]&0xf0)>>4)+'0');
  uart_putc((buffer[k]&0x0f)+'0');
}
for (k=0;k<512;k++) buffer[k]=(0xAA); // 'A'
sd_raw_write(0x00,buffer,512);
sd_raw_read(0x00,buffer,512);       // erase and read first cluster
for (k=0;k<512;k++)
 {uart_putc(((buffer[k]&0xf0)>>4)+'0');
  uart_putc((buffer[k]&0x0f)+'0');
 }
sd_raw_read(512,buffer,512);        // erase and read first cluster
for (k=0;k<512;k++)
 {uart_putc(((buffer[k]&0xf0)>>4)+'0');
  uart_putc((buffer[k]&0x0f)+'0');
 }
for (k=0;k<512;k++) buffer[k]=(0xBB);    // 'B'
sd_raw_write(512,buffer,512);
sd_raw_read(512,buffer,512);        // erase and read first cluster
for (k=0;k<512;k++)
 {uart_putc(((buffer[k]&0xf0)>>4)+'0');
  uart_putc((buffer[k]&0x0f)+'0');
 }
for (k=0;k<512;k++) buffer[k]=(k&0xff);
sd_raw_write(0x00,buffer,512);
for (k=0;k<512;k++) buffer[k]=(0);
sd_raw_read(0x00,buffer,512);
for (k=0;k<512;k++)
 {uart_putc(((buffer[k]&0xf0)>>4)+'0');
  uart_putc((buffer[k]&0x0f)+'0');
}
while (1) {}                        // end of jmf test
```



Ces modifications étant faites, `make && make flash` programme le microcontrôleur et le résultat obtenu est

```
$ cat < /dev/ttyACM0
MMC/SD initialization success
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
41414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141
414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::4242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;000102030405060708090:0;0<0=0>0?101112131415161718191:1;1<1=1>1?2021222324252627
28292:2;2<2=2>2?303132333435363738393:3;3<3=3>3?404142434445464748494:4;4<4=4>4?505152535455565758595:5;5<5=5>5?6061
62636465666768696:6;6<6=6>6?707172737475767778797:7;7<7=7>7?808182838485868788898:8;8<8=8>8?909192939495969798999:9;
9<9=9>9?:0:1:2:3:4:5:6:7:8:9:::;:<:=:>:?;0;1;2;3;4;5;6;7;8;9;:;;;<;=;>;?<0<1<2<3<4<5<6<7<8<9<:<;<<<=<><?=0=1=2=3=4=5
=6=7=8=9=:=;=<===>=?>0>1>2>3>4>5>6>7>8>9>:>;><>=>>>??0?1?2?3?4?5?6?7?8?9?:?;?<?=?>??000102030405060708090:0;0<0=0>0?
101112131415161718191:1;1<1=1>1?202122232425262728292:2;2<2=2>2?303132333435363738393:3;3<3=3>3?40414243444546474849
4:4;4<4=4>4?505152535455565758595:5;5<5=5>5?606162636465666768696:6;6<6=6>6?707172737475767778797:7;7<7=7>7?80818283
8485868788898:8;8<8=8>8?909192939495969798999:9;9<9=9>9?:0:1:2:3:4:5:6:7:8:9:::;:<:=:>:?;0;1;2;3;4;5;6;7;8;9;:;;;<;=
;>;?<0<1<2<3<4<5<6<7<8<9<:<;<<<=<><?=0=1=2=3=4=5=6=7=8=9=:=;=<===>=?>0>1>2>3>4>5>6>7>8>9>:>;><>=>>>??
```

Analyser cette séquence d'échanges et en déduire si la communication avec la carte SD est fonctionnelle.

Si nous "oublions" de connecter une carte SD à la carte Olimex :

```
$ cat < /dev/ttyACM0
MMC/SD initialization failed
MMC/SD initialization failed
```

Nous vérifions ainsi que nous sommes bien en train d'interagir avec la carte SD.

En cas d'échec de la transaction, il peut être judicieux de s'assurer de la taille de la carte SD. En effet, le protocole d'initialisation diffère légèrement selon que les cartes soient de format "classique" (compatible MultiMediaCard – MMC – soit moins de 2 GB) ou de Haute Capacité (SDHC, soit plus de 4 GB). Dans le cas d'une carte de haute capacité, nous prendrons soin d'activer l'option de la bibliothèque `#define SD_RAW_SDHC 1` dans le fichier `sd_raw_config.h`.
Analyser la séquence d'initialisation et comprendre les diverses commandes qui servent à passer la carte SD de son mode natif au mode SPI (CMD0 puis CMD1), et le cas des cartes haute capacité qui nécessitent une initialisation par ACMD41 (ACMD signifie que la commande 41 est préfixée de la requête CMD55 pour annoncer l'utilisation d'une commande de carte SD et non de MMC). On notera en particulier que la documentation des cartes MMC et SD utilise la notation décimale, et non la base 16, pour définir ses commandes.

5.3 Mode interruption

Identifier l'adresse contenant le pointeur vers le gestionnaire d'interruption lié à la fin de transfert d'une transmission SPI.
Identifier le nom du gestionnaire d'interruption requis par `avr-gcc` pour la déclarer.

L'inconvénient de souder la carte SD sur un câble se terminant par un connecteur HE10 est que nous ne pouvons pas extraire la carte SD pour en relire le contenu depuis un PC. Afin de démontrer la capacité à stocker des informations dans une arborescence de type répertoires et fichiers, le format FAT permet une organisation simple et rapides des données. FAT ayant été créé à une époque où les ordinateurs personnels proposaient une puissance de calcul de l'ordre de celle des microcontrôleur actuel, il s'agit du format idéal pour cette tâche. On notera que stocker une grandeur physique toutes les secondes dans un format de 8 bits/donnée sur une carte de capacité de 4 GB permet de poursuivre l'enregistrement pendant ... plus d'un siècle! Une organisation en fichiers classés par date est donc souhaitable (Fig. 7).
Constater que la bibliothèque d'accès aux cartes MMC/SD n'utilise pas l'interruption SPI. Quel serait le gain de l'activer?

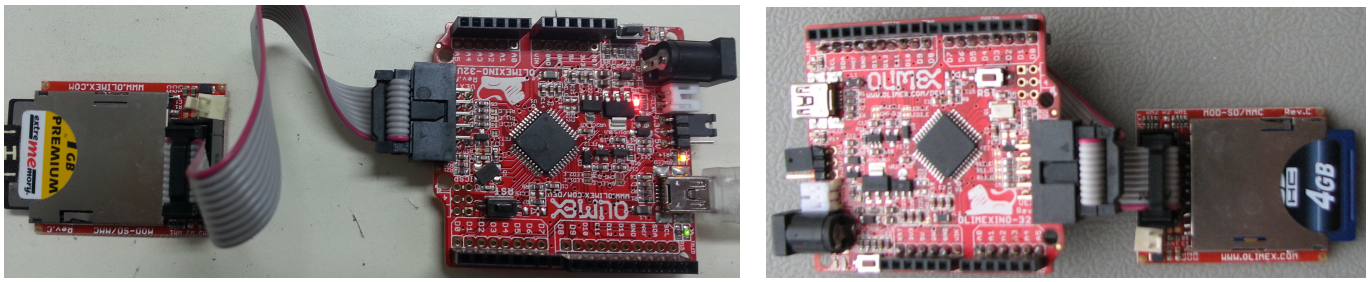


FIGURE 7 – Adaptateur UEXT-carte SD commercialisé par Olimex pour séparer la carte du microcontrôleur après enregistrement des données et ainsi les restituer depuis le lecteur de carte SD d'un PC. Gauche : adaptateur muni d'une carte compatible MMC. Droite : adaptateur muni d'une carte SD HC, nécessitant d'activer `SD_RAW_SDHC`.

Initialiser les fonctions de stockage au format FAT et démontrer la capacité à écrire et lire des données dans un format compréhensible par un PC moderne depuis le microcontrôleur.

Afin de faciliter la tâche et ne pas s'handicaper avec une interface utilisateur, nous nous inspirerons de `main.c` dans `sd-reader_source_20090330` mais nous contentons d'initialiser les liaisons USB et SPI (toujours en faisant appel aux fonctions fournies dans `uart.c`), puis ouvrons le support de stockage (`partition_open(sd_raw_read, sd_raw_read_interval, sd_raw_write, sd_raw_write_interval, 0);`). Ayant validé que lorganisation du support est compris par la bibliothèque, nous ouvrons le système de fichiers (`fat_open(partition);`) et finalement accédons au sommet de l'arborescence (`fat_get_dir_entry_of_path(fs, "/", &directory);`). Pour afficher le contenu d'un fichier, nous nous inspirons de la fonction `cat` du programme d'exemple :

```
1  struct fat_file_struct* fd = open_file_in_dir(fs, dd, "toto");
2  uint8_t buffer[8],i;
3  uint8_t size;
4  uint32_t offset = 0;
5  if(!fd) {uart_puts_p(PSTR("error␣opening\n"));continue;}
6  while((size = fat_read_file(fd, buffer, sizeof(buffer))) > 0)
7    {uart_putdw_hex(offset);
8     uart_putc(':');
9     for (i=0; i<8; ++i) {uart_putc('␣');uart_putc_hex(buffer[i]);}
10    uart_putc('␣');
11 }
12 fat_close_file(fd);
```

Afin d'obtenir ce résultat, nous initialisons la carte SD sous GNU/Linux :

1. `fdisk` pour partitionner la carte SD. Nous créons une unique partition primaire de 64 MB par `fdisk /dev/sdb` ou `fdisk /dev/mmcblk0` puis `n p 1 [valeur par défaut] +64M`. Ensuite, nous informons du type de cette partitions : `t 6` pour fournir le type FAT16. Les informations sont sauvergardées par `w`,
2. formater cette nouvelle partition par `mkfs -t msdos -F 16 /dev/sdb1` ou `mkfs -t msdos -F 16 /dev/mmcblk0p1`
3. monter la carte par `mount /dev/sdb1 /mnt` ou `mount /dev/mmcblk0p1 /mnt` et y créer un fichier `toto` contenant la phrase `hello world` par `echo "hello world" > /mnt/toto`. Démonter la carte (`umount /mnt`) et l'insérer dans le support de carte SD connecté à l'Atmega32U4.

Lancer `minicom`, réinitialiser la carte SD et observer ...

```
MMC/SD initialization success
manuf:  0x74
oem:    JE
prod:   SDC
rev:    10
serial: 0x49918c68
date:   3/14
size:   3807MB
```

```
copy:   0
wr.pr.: 0/0
format: 0
free:   66938880/66959360
toto                                  12
00000000: 68 65 6c 6c 6f 20 77 6f hello wo
00000008: 72 6c 64 0a 6f 20 77 6f rld.
```

En plus d'utiliser les fonction `partition_open()`, `fat_open()`, `fat_open_dir` pour afficher la liste des fichiers (ici `toto` de taille 12 octets) et finalement `open_fil_in_dir()` pour en afficher le contenu, nous avons affiché les propriétés de l'espace de stocage par `print_disk_info(fs);`. Pour un affichage sur USB, on pensera à saupoudrer quelques `CDC_Device_USBTask(&VirtualSerial_CDC_Interface);USB_USBTask();` après chaque `uart_puts_p()` critique au déverminage du code en cours de développement. Ce n'est que lors de l'exécution des ces fonctions de traitement du flux USB que les messages sont réellement affichés : placer la gestion d'USB uniquement dans la boucle `while` infinie ne permet pas d'identifier des dysfonctionnements au cours de l'initialisation (carte mal formatée) ou de l'ouverture du fichier (erreur sur le nom par exemple).

6 Communication I^{2}C

La mise en pratique de la communication I^{2}C se fait sur une horloge temps-réel (*Real Time Clock* – RTC), un périphérique de faible consommation qui permet de réveiller un microcontrôleur de son mode veille profonde dans lequel il consomme le moins d'énergie. Nous utiliserons, pour communiquer sur le bus I^{2}C, les fonctionnalités à cet effet fournies dans `LUFA`.

Consulter `include/libi2c.h` pour identifier les fonctions disponibles, et `src/libi2c.c` pour en appréhender les implémentations.

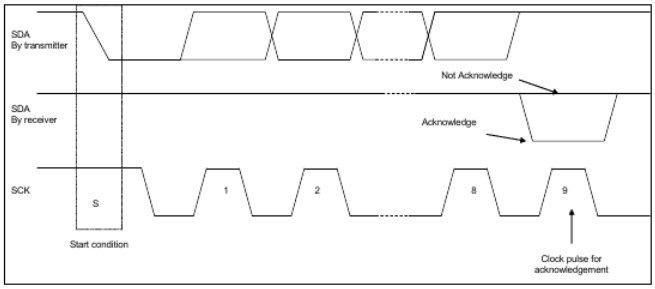


Figure 20: Waveform diagram for I²C acknowledgement on SDA

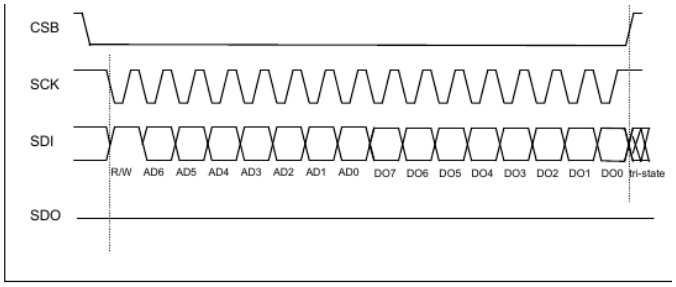


Figure 18: 3-wire SPI read protocol

FIGURE 8 – Extrait de la documentation de l'accéléromètre Bosch BMA220 illustrant les chronogrammes des communications synchrones SPI et I^{2}C. Noter la différence dans l'organisation des données, à prendre en compte lors de l'observation sur oscilloscope des signaux.

On notera quelques subtiles différences entre SPI et I^{2}C. Dans la documentation de l'accéléromètre Bosch BMA220 qui supporte les deux modes de communication (Fig. 8), nous constatons que l'ordre des bits n'est pas le même, un point à mémoriser lors de l'observation des signaux de données à l'oscilloscope. Par ailleurs, I^{2}C adresse les périphériques. contrairement à SPI qui sélectionne l'esclave actif en abaissant un signal de sélection (*Chip Select#*). Par ailleurs, le 8ème bit de la transaction I^{2}C indique la nature de l'échange : 1 pour une lecture, 0 pour une écriture.

Important:
The default slave address assigned to the BMA220 is *000 1011*. When in I²C mode, the LSB can be inverted by tying the CSB pin to '1'. This allows resolving conflicts with existing devices. Also, the 4 LSB can be configured via OTP.

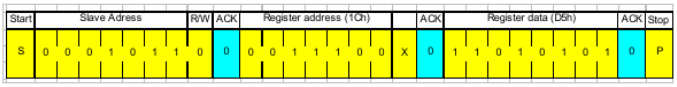


Figure 22: I²C one byte write protocol

Exercice : en observant l'extrait de la documentation de la Fig. 6, quelle adresse faut-il fournir pour lire un registre de l'accéléromètre BMA220 si CSB est connecté à la tension d'alimentation ?

Nous allons nous intéresser à la RTC PCF8563 qui communique avec le microcontrôleur sur un bus I^{2}C. Son adresse est fournie dans la documentation technique. En mode normal, l'horloge est incrémentée toute les secondes. Dans ce mode, les registres de contrôle sont à zéro (par défaut, tous les registres sont initialisés à zéro à la mise sous tension – voir appendice A). Nous nous proposons donc d'initialiser la RTC, et de vérifier que les divers registres évoluent comme prévu dans une horloge, notamment avec les secondes qui s'incrémentent toutes les secondes et les minutes toutes les 60 secondes.

Le programme ci-dessous propose les fonctionnalités minimales pour accéder à la RTC sur bus I^{2}C. Il permet en particulier de vérifier que les divers registres (Fig. 9) contiennent bien les informations annoncées, et que nous avons compris le protocole de communication avec l'horloge (annexe A). Son affichage est cependant fort peu satisfaisant.

Table 4. Formatted registers overview

Bit positions labelled as x are not relevant. Bit positions labelled with N should always be written with logic 0; if read they could be either logic 0 or logic 1. After reset, all registers are set according to Table 27.

Address	Register name	Bit							
		7	6	5	4	3	2	1	0
Control and status registers									
00h	Control_status_1	TEST1	N	STOP	N	TESTC	N	N	N
01h	Control_status_2	N	N	N	TI_TP	AF	TF	AIE	TIE
Time and date registers									
02h	VL_seconds	VL	SECONDS (0 to 59)						
03h	Minutes	x	MINUTES (0 to 59)						
04h	Hours	x	x	HOURS (0 to 23)					
05h	Days	x	x	DAYS (1 to 31)					
06h	Weekdays	x	x	x	x	x	WEEKDAYS (0 to 6)		
07h	Century_months	C	x	x	MONTHS (1 to 12)				
08h	Years	YEARS (0 to 99)							
Alarm registers									
09h	Minute_alarm	AE_M	MINUTE_ALARM (0 to 59)						
0Ah	Hour_alarm	AE_H	x	HOUR_ALARM (0 to 23)					
0Bh	Day_alarm	AE_D	x	DAY_ALARM (1 to 31)					
0Ch	Weekday_alarm	AE_W	x	x	x	x	WEEKDAY_ALARM (0 to 6)		
CLKOUT control register									
0Dh	CLKOUT_control	FE	x	x	x	x	x	FD[1:0]	
Timer registers									
0Eh	Timer_control	TE	x	x	x	x	x	TD[1:0]	
0Fh	Timer	TIMER[7:0]							

FIGURE 9 – Registres de la RTC – vérifier la cohérence du programme avec les informations fournies dans ce tableau.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 16000000UL
#include <util/delay.h>
#include "libttycom.h"
#include "libi2c.h"

#include <string.h>
#include <stdio.h>

#include "VirtualSerial.h"

extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;

int main(void){
  uint8_t i;
  uint8_t mask[]={0x7F, 0x7F, 0x3F, 0x3F, 0x07, 0x1F, 0xFF};
  uint8_t time[16];               // Le PCF8563 possede 16 registres
  char buffer[80]="C'est␣parti\r\n\0";
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();
  fputs(buffer, &USBSerialStream);
  I2C_UEXT(ON);                   // Alimentation du connecteur UEXT on (D8=PB4)
  I2C_init(24, 0);                // Initialisation du bus I2C : 24 pour 400kHz
  I2C_stop();

  time[0]=2;
  time[1]=1; time[2]=2; time[3]=3; time[4]=4; time[5]=5; time[6]=6; time[7]=9;
  // second    min         hour       day      weekday     month     year

  // initialisation RTC
  I2C_write(PCF8563,(uint8_t *)time,8);   // tester l'echec
```

```
35
36   while (1){
37     time[0]=2;                              // registre qu'on veut lire
38     I2C_rw(PCF8563, time, 1, time+1, 7);  // lecture de la RTC
39     for(i=0; i<7; i++)
40        {time[i+1]=time[i+1]&mask[i];
41         buffer[2*i+1]=(time[i+1]&0x0f)+'0';
42         buffer[2*i]=((time[i+1]&0xf0)>>4)+'0';
43        }
44     buffer[14]='\r';buffer[15]='\n';buffer[16]=0; // affichage
45     fputs(buffer, &USBSerialStream);
46     CDC_Device_ReceiveByte(&VirtualSerial_CDC_Interface);
47     CDC_Device_USBTask(&VirtualSerial_CDC_Interface);
48     USB_USBTask();
49     _delay_ms(4000);                        // attente pour laisser RTC tourner
50   }
51   return 1;
52 }
```

1. **Expliquer l'utilisation du tableau `mask` dans l'affichage.**
2. **Proposer une fonction qui convertit l'affichage du contenu des registres proposé ici (`valeur + '0'`) par une fonction qui affiche le contenu de l'horloge par un utilisateur humain. On prendra en particulier soin de noter que la RTC stocke les informations en BCD.**
3. **Proposer une fonction qui remplace l'initialisation statique proposée dans cet exemple par une demande à l'utilisateur de fournir date et heure, convertisse cette information (ASCII) en format exploitable par la RTC (BCD), et initialise la RTC en conséquent.**

6.1 Alarme de la RTC pour générer une interruption sur Atmega32U4

La RTC propose un mode alarme dans lequel le statut courant de l'horloge (minutes, heures ...) et comparé avec une valeur pré-définie (date de l'alarme). S'il y a coïncidence, un signal actif au niveau bas se déclenche sur la broche INT#, connectée à la broche 4 du connecteur UEXT (D0 (RXD)) qui est liée à D0 de la carte Olimex (INT2).

Pour utiliser l'alarme, il est nécessaire de positionner les bits suivants :

— `AEI` dans le registre 1 à 1 autorise l'interruption sur l'alarme,

— les bits `AE_x` (`x= M, H D` et `W`) à "1" *désactivent* la comparaison de l'alarme avec l'horloge,

— si le bit `AF` est à 1, ce drapeau indique l'égalité de l'alarme et de l'horloge. Il doit être remis à zéro après interruption.

Déclencher une interruption sur l'ATMega 32U4 en notant que le signal – actif au niveau bas – issu de l'alarme est connecté à la broche D0 (INT2) du microprocesseur. On pourra tester manuellement le bon fonctionnement de l'interruption matérielle en reliant par un fil D0 et la masse. Afin de ne pas attendre une minute le déclenchement de l'interruption, on pensera à initialiser l'horloge avec une valeur des secondes proche de 60.

6.2 Mode interruption de la RTC

Le schéma du module d'horloge temps-réelle est disponible à `https://www.olimex.com/Products/Modules/Time/MOD-RTC/resources/MOD-RTC_Rev_B.pdf`

Nous y découvrons en particulier que ...

⚠... les résistances R4 (330 Ω) et R8 (2200 Ω) pour le tirage à la tension d'alimentation ne sont pas peuplées sur la carte commercialisée par Olimex. L'activation de l'interruption pour réveiller le microcontrôleur nécessite de souder ces deux composants passifs (Fig. 10).

Une fois la résistance reliant INT# du module RTC à INT0 du microcontrôleur mise en place – on notera que cette broche sert aussi à la liaison du PC vers le microcontrôleur en RS232, qu'on prendra soin de désactiver – le programme ci-dessous met en veille le microcontrôleur et le réveille chaque minute. On notera en particulier l'acquittement de l'interruption, en communiquant sur le bus I2C dans le gestionnaire d'interruption, afin d'abaisser la broche INT# et éviter de boucler dans le gestionnaire d'interruption. Une fois l'interruption acquittée, un drapeau est mis en place et la prochaine alarme est programmée dans le programme principal.

```
1 #include <avr/io.h>
2 #include <avr/interrupt.h>
3 #include <avr/sleep.h>
4 #include <avr/wdt.h>
5 #define f_CPU 16000000UL
6 #include <util/delay.h>
7 #include "libttycom.h"
8 #include "libi2c.h"
```

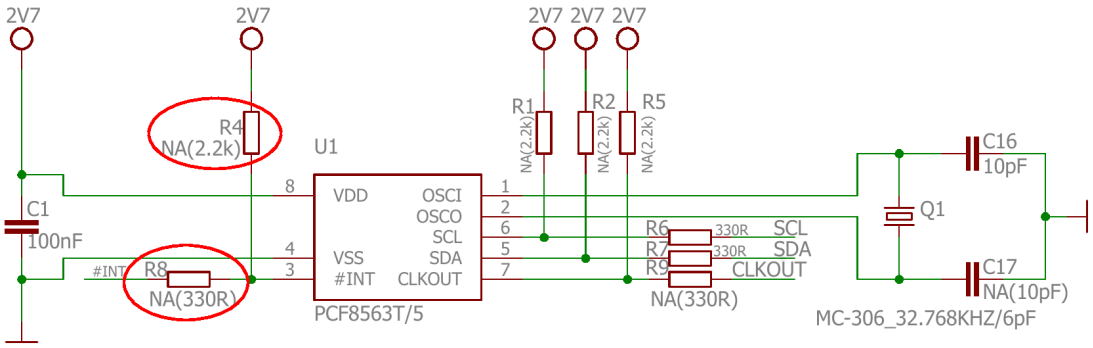


FIGURE 10 – Composants manquant pour faire fonctionner le réveil du microcontrôleur sur déclenchement d'interruption par la RTC.

```
9
10 #include <string.h>
11 #include <stdio.h>
12
13 #define USART_BAUDRATE 9600
14
15 volatile int alarme=0;
16
17 // https://github.com/tomvdb/avr_arduino_leonardo/blob/master/examples/uart/main.c
18 void uart_transmit(char data )
19 {while (!(UCSR1A&(1<<UDRE1))) ;
20  UDR1 = data;
21 }
22
23 void uart_puts(char *c)
24 {int k=0;
25  while (c[k]!=0) {uart_transmit(c[k]);k++;}
26 }
27
28 void uart_init()
29 {unsigned short baud;
30  UCSR1A = 0;                          // importantly U2X1 = 0
31  UCSR1B = 0;
32  UCSR1B = (1 << RXEN1)|(1 << TXEN1); // enable receiver and transmitter
33  UCSR1C = _BV(UCSZ11) | _BV(UCSZ10); // 8N1
34  // UCSR1D = 0;                          // no rtc/cts (probleme de version de libavr)
35  baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
36  UBRR1H = (unsigned char)(baud>>8);
37  UBRR1L = (unsigned char)baud;
38 }
39
40 void enterSleep(void)
41 {set_sleep_mode(SLEEP_MODE_IDLE);
42  sleep_enable();
43  sleep_mode(); //enter sleep mode ... good night
44  sleep_disable(); //wake up from interrupt
45 }
46
47 ISR(INT2_vect) //front descendant
48 {alarme=1;
49  PORTB^=1<<PORTB5;
50  _delay_ms(100);
51  }
52
53 int main (void)
54 {int8_t dummy;
55  uint8_t i;                                          //            m_alm          week_alm
56  uint8_t mask[13]={0xC8,0x1F,0x7F, 0x7F, 0x3F, 0x3F, 0x07, 0x1F, 0xFF, 0xFF,0xBF,0xBF,0x87};
57  uint8_t time[16]; //le PCF8563 possede 16 registres
58
59  wdt_disable();
```

```
 uart_init();

 MCUCR &=~1<<PUD;
 DDRD&=~1<<PORTD2;
 PORTD|=1<<PORTD2;

 I2C_UEXT(ON); //alimentation du connecteur UEXT on (D8=PB4)
 I2C_init(24,0); //initialisation du bus I2C : 24 pour 400kHz
 I2C_stop();

 time[0]=1; //adresse du premier registre (ici seconde)
 time[1]=0x02;
 time[2]=0x55; time[3]=01; time[4]=05; time[5]=06; time[6]=7; time[7]=8; time[8]=9;
//valeur des registres a partir duquel on veut imposer la valeur
//   second      min         hour         day       weekday      month      year

 I2C_write(PCF8563, (uint8_t *) time, 9); // initialisation RTC
 // @ PCF 8563, puis @ premier registre (time[0]) puis val des 8 premiers registres (time[1] a time[7])

 // initialisation alarme
 time[0]=9; //adresse du premier registre (ici seconde)
 time[1]=0x02; time[2]=0x82; time[3]=0x83; time[4]=0x84;

 I2C_write(PCF8563, (uint8_t *) time, 5); //tester l'echec

 EIMSK = 1<<INT2; //enable int2
 EICRA=1<<ISC21; //front descendant
 USBCON=0;
 sei();

 while(1)
  {time[0]=1;
   I2C_rw(PCF8563, time, 1, time+1, 12); //lecteur de la RTC //8

   for(i=1; i<=12; i++)
     {time[i]=time[i]&mask[i];
      buffer[2*i-1]=(time[i]&0x0f)+'0';      if (buffer[2*i-1]>'9') buffer[2*i-1]=buffer[2*i-1]-'9'+'A';
      buffer[2*i-2]=((time[i]&0xf0)>>4)+'0';if (buffer[2*i-2]>'9') buffer[2*i-2]=buffer[2*i-2]-'9'+'A';
     }
    buffer[2*12]=0;
    uart_puts(buffer);

    if (alarme==1)
      {sprintf(buffer,"␣ALARME");uart_puts(buffer);
       time[0]=9;time[1]=time[3]+1; // status=1 seconde=2 minute=3
       I2C_write(PCF8563, (uint8_t *) time, 2);
       time[0]=1;time[1]=0x02;
       I2C_write(PCF8563, (uint8_t *) time, 2);
       alarme=0;
       PORTE ^=1<<PORTE6; //toggle led
       enterSleep();
     }
   uart_puts("\r\n\0");
   _delay_ms(1000); //attente pour laisser RTC tourner
  }
  return 0;
}
```

7 Écran LCD

Les écrans de téléphone portable Nokia communiquent par lien synchrone s'apparentant au SPI.

Cependant, les données sont codées sur 9 bits, avec le premier bit indiquant si la transmission concerne une commande ou une donnée. L'Atmega32U4 ne supporte pas de transmissions sur SPI en 9 bits : l'opportunité s'offre donc à nous d'implémenter SPI de façon logicielle.

```
#define select_card PORTB &= ~(1 << PORTB5)
```

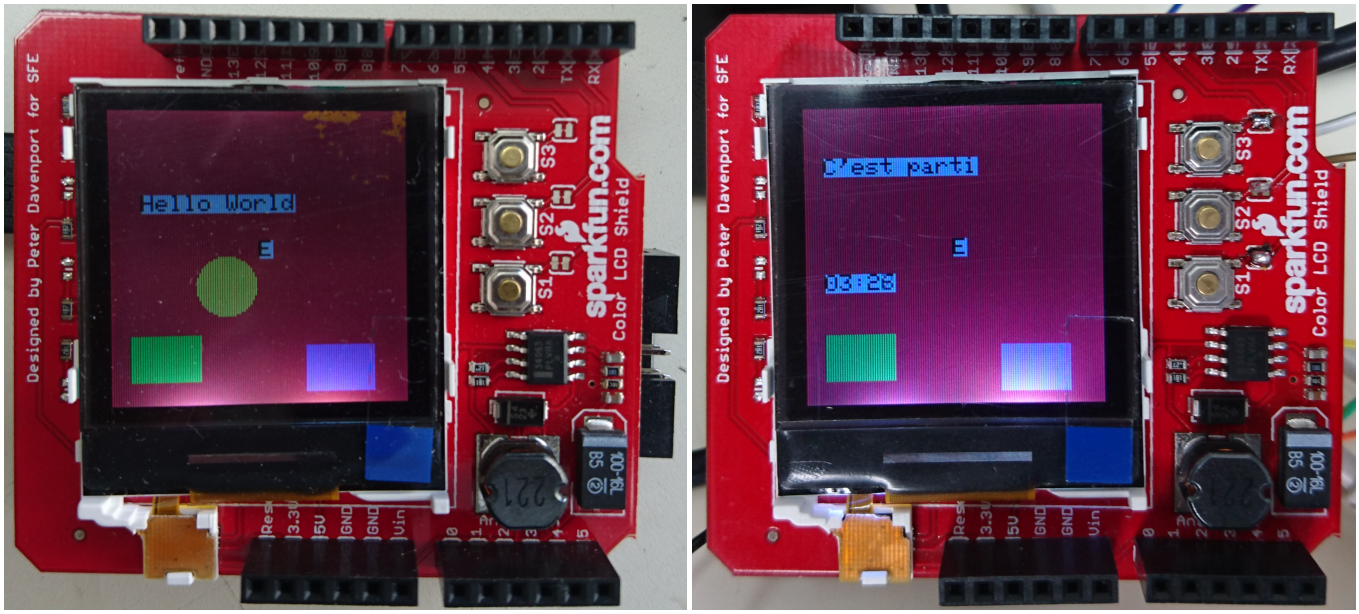


FIGURE 11 – Affichage sur écran LCD : à gauche un message statique et quelques motifs géométriques ; à droite la date (minutes :secondes) lue sur RTC par I^2C.

```
#define unselect_card PORTB |= (1 << PORTB5)

#define mosi_up   PORTB |= (1 << PORTB7)
#define mosi_down PORTB &= ~(1 << PORTB7)
#define ck_up     PORTC |= (1 << PORTC7)
#define ck_down   PORTC &= ~(1 << PORTC7)

void sendByte(bool cmd, u8 data)
{int k;
 ck_up;
 select_card;
 if (cmd==0) mosi_down; else mosi_up;
 ck_down;attend();ck_up;attend();
 for (k=7;k>=0;k--)
   {if (((data>>k)&0x01) !=0) mosi_up; else mosi_down;
    ck_down;attend();ck_up;attend();
   }
 unselect_card;
}
```

La seule subtilité lors de l'interfaçage à la fois du LCD et du RTC est que la même broche sert à la fois à activer le port UEXT et la réinitialisation de l'écran.

Ce programme met en évidence une limitation de l'architecture Harvard de l'Atmega32U4. Classiquement, sur un processeur mélangeant mémoire dédiée aux données et aux instructions, le préfixe `const` informe le compilateur de la capacité à stocker une donnée statique en mémoire non-volatile, dans cet exemple les polices de caractères. Ayant beaucoup plus de mémoire non-volatile que de mémoire volatile (RAM), cette dernière est libérée pour laisser la place aux variables manipulées au cours de l'exécution du programme. Une architectre Harvard dédie un bus et une mémoire volatile aux données, et un autre bus et une mémoire non-volatile aux instructions. Dans ce contexte, le compilateur est incapable de stocker les tableaux des polices de caractères en mémoire non-volatile, et l'intégralité de la mémoire volatile est occupée par ces tableaux. En pratique, la quantité de mémoire volatile est insuffisante et le programme crashe par corruption de la mémoire lors de l'accès aux polices de caractères. Nous devons donc prendre soin de ne charger qu'une police.

A Annexe

Extrait de la *datasheet* de la RTC qui indique que le premier octet transmis sur le bus I^{2}C contient l'adresse du premier registre à remplir.

9.5.1 Addressing

Before any data is transmitted on the I^{2}C-bus, the device which should respond is addressed first. The addressing is always carried out with the first byte transmitted after the start procedure.

The PCF8563 acts as a slave receiver or slave transmitter. Therefore the clock signal SCL is only an input signal, but the data signal SDA is a bidirectional line.

Two slave addresses are reserved for the PCF8563:

Read: A3h (10100011)

Write: A2h (10100010)

The PCF8563 slave address is illustrated in Figure 18.

Fig 18. Slave address

9.5.2 Clock and calendar READ or WRITE cycles

The I^{2}C-bus configuration for the different PCF8563 READ and WRITE cycles is shown in Figure 19, Figure 20 and Figure 21. The register address is a 4-bit value that defines which register is to be accessed next. The upper four bits of the register address are not used.

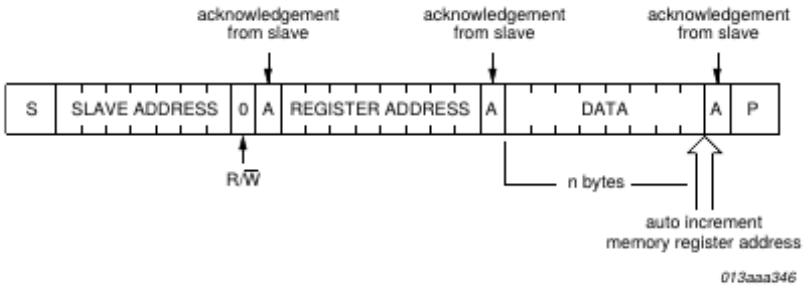


Fig 19. Master transmits to slave receiver (WRITE mode)

Références

[1] J.-M Friedt, É. Carry, *Enregistrement de trames GPS – développement sur microcontroleur 8051/8052 sous GNU/Linux*, GNU/Linux Magazine France, **81** (Février 2006)