Mise en œuvre d'un système de contrôle réparti sur un réseau hétérogène

P. Gaussier, Ph. Laroque

TP Master Pro SIIC – octobre 2006

1 Annexes

1.1 Le Basic Stamp

1.1.1 Instructions du Basic Stamp utiles pour les TPs

Types de données:

- BIT (valeurs 0 ou 1)
- NIB (valeurs entre 0 et 15 - hexa)
- BYTE (valeur entre 0 et 255 - octet)
- WORD (valeur entre 0 et 65535 - entier court)

déclarations mots réservés pour déclarer une variable est VAR et CON pour des constantes (qui sont directement remplacées à la compilation par leur valeur constante).

création d'un tableau chaine VAR BYTE(10) 'crée un tableau de 10 octets ou 10 caractères l'apostrophe permet d'indiquer un commentaire (non transmis au BS)

références On peut créer plusieurs variables qui font référence à une même variable:

```
x VAR BYTE
z VAR x
```

z est équivalent à x. Les 2 noms font référence à la même zone mémoire.
De manière plus intéressante, on peut créer des variables accédant à une partie d'une autre variable. Par exemple pour récupérer l'octet de poids fort et l'octet de poids faible de x dans 2 variables de nom différent mais correspondant toujours à notre variable x, il suffit d'écrire:

```
x VAR BYTE
tete VAR x.HIGHBYTE 'ref aux 8 bits de poids fort
queue VAR x.LOWBYTE 'ref 8 bits de poids faibles
```

On peut faire de même pour récupérer le 1er bit de l'octet de poids fort:

```
z VAR x.HIGHBYTE.LOWNIB.BIT1
```

`LOWNIB` correspond au premier des 2 chiffres en hexa permettant de coder l'octet (`HIGHNIB` pour le 2eme). Sur un mot binaire de 2 octets (entier court/ `WORD`), il est possible d'accéder à n'importe le quel de ces bits directement en combinant (`LOWBYTE`, `HIGHBYTE`, `BYTE0`, `BYTE1`, `LOWNIB`, `HIGHNIB`, `NIB0`, `NIB1`, `NIB2`, `NIB3`, `LOWBIT`, `HIGHBIT`, `BIT0`, `BIT1` ...`BIT15`). L'emploie de ces symboles se fait sur tous les types de données du PBASIC dans la limite du lié à chaque type.

Instructions de branchement et boucles

- IF test THEN adresse (attention, il n'y a pas de else et on ne peut pas exécuter une instruction en même temps. Il faut systématiquement faire un saut à un label (comme si l'on utilisait des goto).
- BRANCH ...
- GOTO ...
- GOSUB ...
- RETURN
- FOR ... NEXT

affichage d'information sur le terminal d'un PC pour debug DEBUG

opérateurs

- +,-,*,/
- << et >> décalage à gauche ou à droite
- & ET logique
- | OU logique
- ^ XOR

principales fonctions mathématiques

- ABS
- COS / SIN
- DCD
- ~ inverse
- SQR racine carrée

gestion et configuration d'entrées/sorties numériques

- INPUT / OUTPUT / REVERSE
- LOW / HIGH / TOGGLE
- PULSIN / PULSOUT
- BUTTON
- COUNT

gestion de liaisons série asynchrones

- SERIN
- SEROUT
- OWIN (2p)
- OWOUT (2p)

Instruction pour gérer des entrées sorties analogiques

- PWM
- RCTIME

Autres instructions

- génération de sons (FREQOUT, DTMFOUT)
- pause pendant un certain nombre milli secondes (PAUSE)
- acces à l'EPROM (DATA, READ, WRITE...)
- accès à la RAM (PUT, GET)
- commande d'un écran LCD (LCDCMD, LCDIN, LCDOUT)
- commande d'une liaison série synchrone I2C (SHIFTIN, SHIFTOUT, I2CIN, I2COUT)
- commande d'une liaison X10 domotique (XOUT)
- gestion d'énergie (NAP, SLEEP, END)

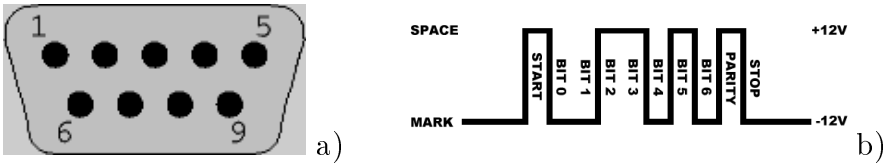

Figure 1: a) Fiche DB9 utilisé pour les liaisons séries. b) Exemple de transmission série asynchrone.

1.2 De la liaison série aux réseaux de terrain

1.2.1 Liaison série RS232

La liaison série RS232 est une liaison asynchrone. Pour envoyer une information à un périphérique, il faut un minimum de 2 fils: 1 fils de masse (référence de potentiel pour l'émetteur et le récepteur) et 1 fil de transmission de données (passant d'un niveau bas à un niveau haut en fonction des données binaires à transmettre). La transmission des données est unidirectionnelle. Si l'on désire que le récepteur puisse renvoyer des données à l'émetteur, il faut introduire un fil supplémentaire. Typiquement, on retrouve dans une liaison série bidirectionnelle un minimum de 3 fils nommé Rx (réception données), Tx (transmission/émission de données) et GND (la masse). Si l'on veut permettre une gestion des flux de données, d'autres lignes liées au "handshaking" doivent être utilisées.
Attention: la liaison doit être établie avant l'alimentation de la carte. La configuration de la ligne doit indiquer les paramètres suivants: 2400 bds, 8 bits, no parity, 1 bit de stop. Enfin, après un ordre "Q", il est nécessaire de remettre la carte sous tension.
Contrairement aux circuits logiques classiques, le niveau logique 1 correspond à -12V et le niveau logique 0 correspond à +12V. Cette caractéristique permet des communications sur de plus longues distances sans nécessiter d'amplificateur. Lorsque l'on décide d'utiliser une patte du basicstamp comme liaison série, il est nécessaire d'introduire en série une résistance de 22Kohm pour limiter la tension dans un domaine acceptable pour le circuit (voir manuel de programmation p 275).
Dans le cas d'une liaison série synchrone, il est nécessaire d'introduire un 3eme fil correspondant au signal d'horloge.

1.2.2 Liaison USB

Contrairement aux liaisons séries classiques, la liaison USB (Universal Serial Bus) permet d'alimenter des périphériques et utilise une transmission des données par paquets. Cette liaison nécessite 4 fils et permet un débit de 12 Mb/s en full-speed et de 1.5 Mb/s en low-speed (selon la bande passante nécessaire). D'autre part, USB permet de dialogue avec plusieurs périphériques (endpoints) caractérisés par un identifiant unique (fixé par le constructeur). Chaque endpoint possède une direction de flux prédéterminée (in/out). Un certain nombre de paramètres permettent de décrire les propriétés du périphérique en terme de communication (temps de latence et fréquence d'accès au bus, bande passante, adresse du endpoint, gestion d'erreurs, taille maximale des paquets qu'il peut recevoir ou envoyer, type de transfert, direction des transferts). On distingue 4 types de transferts: transfert de contrôle (ponctuels et à l'initiative de l'hôte utilisé pour les opérations de configuration et de contrôle de statut), transfert synchrone (périodique, continu utilisé pour les webcams par ex.), transfert d'interruption (données de petite taille, faible temps de latence, faible fréquence comme les souries USB par ex.), bulk transfert (large quantités de données, non périodique et ne nécessitant pas une bande de passante prédéterminée / les scanners par ex.).

Il existe des circuit permettant l'interfaçage direct entre un bus USB et une liaison série classique (transparents pour l'utilisateur et le développeur).

1.2.3 Le réseau local industriel CAN

Les liaisons ethernet classiques sont très sensibles aux interférences électromagnétiques. Elles ne fonctionnent pas très bien dans des environnements difficiles tels que les usines (présence de machines électriques), les voitures, bateaux... Le réseau CAN (Controller Area Network) propose une solution unifiée pour les communications séries dans ce type d'environnement. Le réseau CAN a été conçu à l'origine par Bosh (norme ISO 11898) pour gérer le nombre sans cesse croissant de processeurs embarqués dans les voitures (voir fig. **??**). Un réseau CAN gère les couches 1 et 2 du modèle OSI. Il permet des connexions du type multipoint par "OU cablé", la transmission synchrone sur paire torsadée, fibre optique ou liaison infrarouge. Il permet un débit maximal de 1Mbit/s sur 40m. L'accès au support se fait par compétition. Un mécanisme de détection d'erreur et de retransmission automatique est géré de même que la gestion de messages prioritaires. Enfin la liaison CAN permet de garantir les délais de transmission pour les applications temps réel.

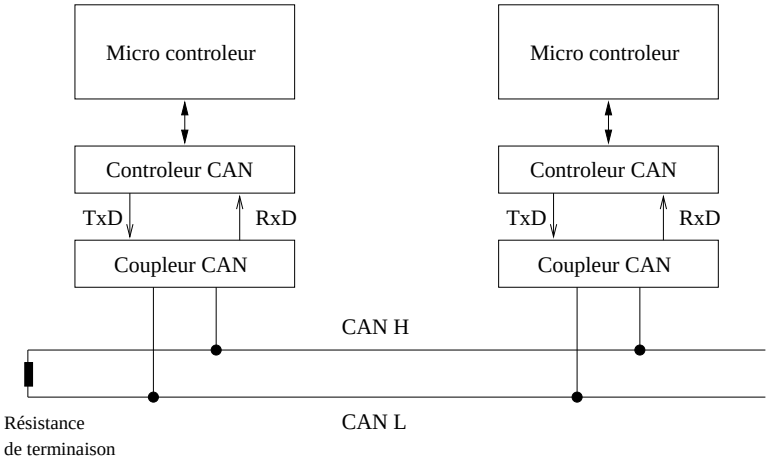

Figure 2: Connexion des organes sur réseau CAN

Afin de traiter les informations en temps réel, et de garantir un délai entre la commande et l'action, il est nécessaire de pouvoir assigner très rapidement le bus en cas de conflit (lorsque plusieurs stations souhaitent transmettre en même temps). Sur le réseau CAN, l'identificateur de chaque message est un mot de 11 bits situé en début de trame (29 bits pour le format étendu) qui détermine sa priorité. Les priorités doivent être assignées lors de l'analyse conceptuelle du réseau (pas de modification dynamique). Le procédé d'attribution du bus utilise un arbitrage "bit à bit". Les noeuds en compétition, émettant simultanément sur le bus, comparent bit à bit l'identificateur de leur message avec ceux concurrents. La station de plus forte priorité gagne.

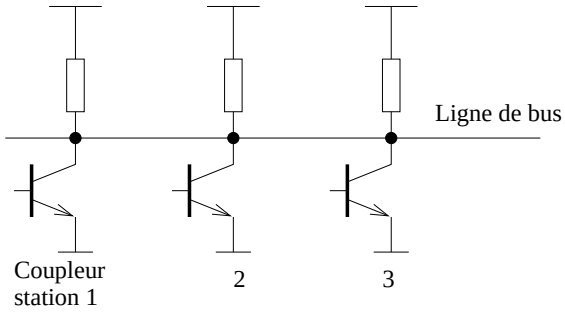

Figure 3: "OU câblé" utilisé pour le contrôle du réseau CAN.

Les stations sont connectées sur le bus par un "OU câblé" (fig. ??). En cas d'émission simultanée, la valeur 0 écrase la valeur 1. L'état 0 est donc appelé état dominant (et 1 l'état récessif). Lors de l'arbitrage bit à bit, dès qu'une station émettrice se trouve en état récessif et détecte un état dominant, elle perd la compétition et arrête d'émettre. Tous les perdants passent en état de réception et ne tentent plus d'émettre tant que le bus est utilisé (voir fig. ??).

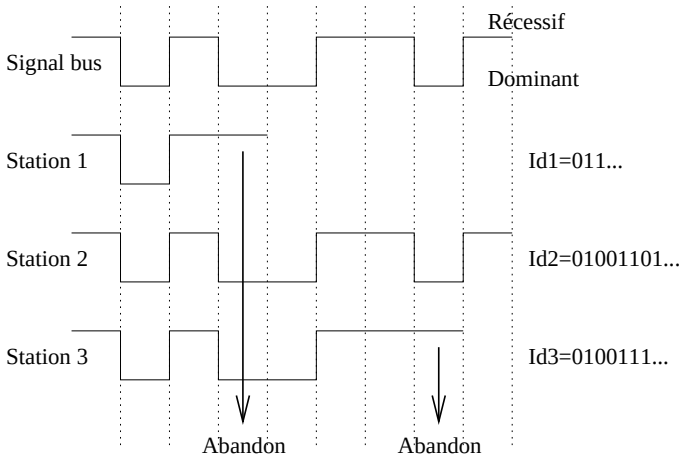

Figure 4: Exemple de l'effet du "OU câblé" lorsque 3 stations veulent émettre simultanément (elles commencent par transmettre leur identifiant).

Une trame d'un réseau CAN est composée des champs suivants:

- bit SOF (Start of Frame)
- identificateur sur 11 bits (utilisé pour l'arbitrage)
- bit RTR (Remote Transmission Request) détermine s'il s'agit d'une trame de données (état dominant) ou d'une trame de demande de message
- bit IDE pour distinguer entre le format standart (état dominant) et le format étendu
- bit réservé pour une utilisation future
- 4 bits DLC (Data Lenght Code) pour indiquer le nombre d'octets contenu dans la zone de données
- champ de données (entre 0 et 8 octets)
- champ CRC de 15 bits
- champ ACK composé d'un bit à l'état récessif (forcé à l'état dominant par les stations ayant bien reçu la trame) suivi d'un bit séparateur ACK
- champ EOF (End of Frame) de 7 bits permettant d'identifer la fin de la trame

D'autres réseaux locaux tels que Modbus ou Jbus sont utilisés pour interconnecter des automates industriels en utilisant un protocole de type maître/esclave. Les connexions séries asynchrone en bande de base sont réalisées sur des connexions multi-points RS485 avec des débits limités à 19.5 kbit/s. Le protocole de niveau 2 associé définit les trames de commande et de réponse utilisées pour lire l'état ou pour commander des automates. Modbus peut en outre être encapsulé dans des trames TCP/IP.

1.3 Exemples de programmes BASIC

Sous linux, les basic stamps peuvent soit se programmer en ligne de commande en utilisant stampbc par exemple soit au travers d'une IHM: gstamp. Ces 2 programmes font appel à une librairie dynamique tokeniser.so qu'il faut penser à mettre dans /usr/lib (si vous êtes super utilisateur!). Dans le cas contraire, vous pouvez rajouter à votre variable d'environnement LD_LIBRARY_PATH le chemin d'accès à la librairie.

gstamp ne fonctionne que si le basic stamp est connecté au premier port série (bug dans la commutation de port). Une fois votre programme télé-chargé sur le BS2 (voir fig. **??**), vous pouvez vous servir de minicom pour transformer votre PC en terminal série du BS2. Vous pouvez alors voir le résultat de votre programme s'afficher sur ce terminal (faire un reset du BS2 pour recommencer l'exécution de votre programme). Vous pourrez aussi envoyer des ordres au BS2 si vous avez prévu de lire la liaison série du BS2.

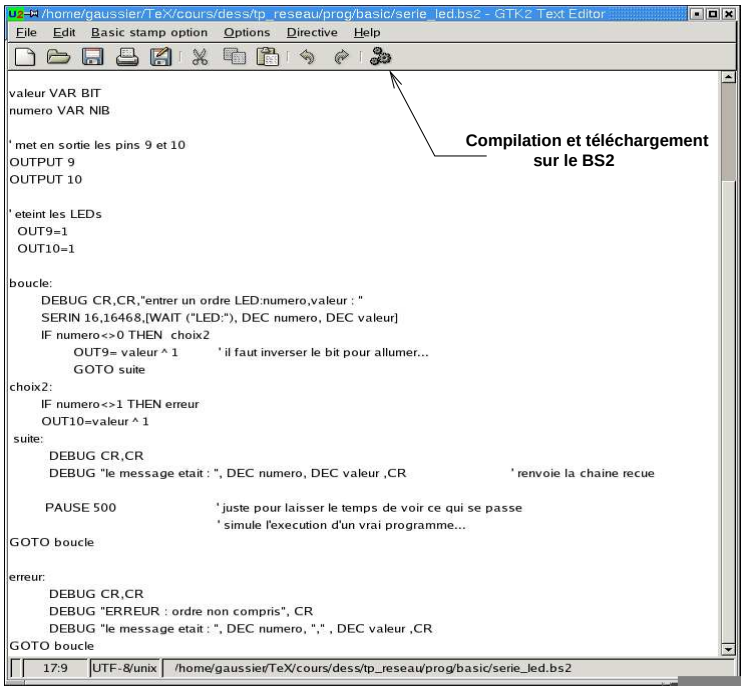

Figure 5: Snapshot de gstamp en cours d'édition d'un programme basic pour le BS2.

Pour que minicom fonctionne correctement mettre "echo off" et configurer la liaison série en 9600-N-1 (9600 Bd, pas de parité, 1 bit de stop) ou une autre vitesse selon votre programme. Vous pouvez aussi vous mettre en mode terminal ansi et demander à ajouter systématiquement un line feed (add line feed).

1.3.1 led.bs2 et free_emission.bs2

Le programme free_emission.bs2 se contente de tester la liaison série en envoyant des messages sur la liaison série en utilisant l'ordre DEBUG (écrit les nombres de 1 à 255).
Le programme led.bs2 configure les pin 9 et 10 en sortie et les positionne au niveau haut pour éteindre les LED qui y sont connectées. Ensuite, le programme passe alternativement les sorties du niveau haut à bas pour allumer et éteindre les LED en fonction d'un timing donné par l'instruction PAUSE. Le programme affiche aussi sur la liaison série un petit message.

```
'{$Stamp bs2}
' free_emission.bs2
' Test emission sur la liaison serie
' P. Gaussier sept. 2003

indice VAR BYTE

' met en sortie les pins 9 et 10
OUTPUT 9
OUTPUT 10

' allume les LEDs
   OUT9=0
   OUT10=0

   DEBUG CR,CR,"essai liaison serie (emission)",CR

boucle:

   FOR indice=1 TO 255
       DEBUG "val=", DEC indice,CR
   NEXT

  GOTO boucle
```

```
'{$Stamp bs2}

' led.bs2
' Commande de 2 LEDs connectees aux pattes 9 et 10 d un basic stamp
' P. Gaussier sept. 2003

' met en sortie les pins 9 et 10
output 9
output 10

' eteint les LEDs
   out9=1
   out10=1

debug "Programme de test:  activation LEDs",CR

loop:

   out9=1: ' allume la LED sur 9
   out10=0: ' etteint la LED sur 10

   pause 500

   out9=0: ' allume la LED sur 9
   out10=1: ' etteint la LED sur 10

   pause 500

' eteint les LEDs
   out9=1
   out10=1

   pause 1000

goto loop
```

1.3.2 serie_test.bs2

Ce programme teste la lecture de la liaison série standard du BasicStamp. Le programme attend une chaîne d'au plus 9 caractères et/ou se terminant par une *. Une fois cette chaîne reçue, elle est affichée. Une LED s'éteint juste après la réception des données et se rallume lorsque le BasicStamp est prêt à traiter de nouvelles données (simule l'exécution d'un long programme pendant lequel la liaison série n'est pas consultée).

ATTENTION:

- Il n'y a pas de buffer hardware sur l'entrée série du BasicStamp.
- Une liaison série ne peut normalement être utilisée que par un seul programme à la fois. Si vous utilisez `minicom`, pour dialoguer manuellement avec le basicstamp, il ne faut pas oublier de sortir de `minicom` pour pouvoir charger un autre programme dans l'EPROM du basicstamp.

```
'{$Stamp bs2}
' serie_test.bs2

' Test de la liaison serie: entree d'en texte 9600Bd No parity 8 bits
' sur le port serie de programmation du Basic Stamp
' P. Gaussier sept. 2003

message VAR BYTE(10)

' met en sortie les pins 9 et 10
OUTPUT 9
OUTPUT 10

' eteint les LEDs
OUT9=1
OUT10=1

boucle:
  OUT9=0: ' allume la LED sur 9

  message(9)=0 ' met 0 dans le dernier caractere
  DEBUG CR,CR,"entrer un message (terminer par *) :  "
  SERIN 16,16468,[STR message\9\"*"]' attend 9 caracteres ou une *

  OUT9=1: ' etteint la LED sur 9
  DEBUG CR,CR
  DEBUG "le message etait :  ", STR message ,CR ' renvoie la chaine recue

  PAUSE 500 ' juste pour laisser le temps de voir ce qui se passe
                                                ' simule l'execution d'un vrai programme...
GOTO boucle
```

1.3.3 Mise en oeuvre de la liaison serie en C : serial.c

Voici l'un des fichiers de bases nécessaire à la gestion de la liasion série en C. Pour utiliser la liaison série, il faut d'abord ouvrir le port série correspondant. Par défaut, la plupart des PCs possèdent 2 ports séries accessibles via /dev/cua0 ou /dev/cua1. Il est aussi possible d'y accéder en utilisant /dev/ttyS0 et /dev/ttyS1. Une couche logicielle supplémentaire gère les propriétés liés aux terminaux.

Dans le programme serial.c, la configuration de la liaison série est réalisée par la fonction serial_open() qui renvoie dans la variable entière serialport un handler sur la liaison série que l'on vient d'ouvrir.

L'envoie et la réception de message au travers d'une liaison série se font avec les instructions classiques write et read. Afin d'être sur les données ne restent pas dans les buffers hardwares et logiciels du PC, on peut utiliser les ordres tcdrain et tcflush (pour vider les buffers). Cela peut se révéler crucial pour une application temps réel.

```
/* fichier : serial.c */

#include <termio.h>
#include <fcntl.h>
#include <string.h>


#define PORT "/dev/ttyS0"
/*#define BAUDRATE B9600*/
#define BAUDRATE B2400 /* vitesse de transmission utilisee */

#include "serial.h"

int serialport;

struct termios oldtio,newtio;

/*------------------*/
/* serial.c */
/* initialisation de la liaison */
/* serie, lecture et ecriture */
/*------------------*/

void serial_open()
{

  int fd,c, res;

/*
  Open modem device for reading and writing and not as controlling tty
  because we don't want to get killed if linenoise sends CTRL-C.
*/
 fd = open(PORT, O_RDWR | O_NOCTTY );
 serialport=fd;
 if (fd <0) {perror(PORT); exit(-1); }

 tcgetattr(fd,&oldtio); /* save current serial port settings */
 bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

/*
  BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
  CRTSCTS : output hardware flow control (only used if the cable has
            all necessary lines. See sect. 7 of Serial-HOWTO)
  CS8 : 8n1 (8bit,no parity,1 stopbit)
  CLOCAL : local connection, no modem contol
  CREAD : enable receiving characters
*/
 newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

/*
  IGNPAR : ignore bytes with parity errors
  ICRNL : map CR to NL (otherwise a CR input on the other computer
```

```
                will not terminate input)
    otherwise make device raw (no other input processing)
*/
 newtio.c_iflag = IGNPAR | ICRNL;

/*
 Raw output.
*/
 newtio.c_oflag = 0;

/*
   ICANON : enable canonical input
   disable all echo functionality, and don't send signals to calling program
*/
 newtio.c_lflag = ICANON;

/*
   initialize all control characters
   default values can be found in /usr/include/termios.h, and are given
   in the comments, but we don't need them here
*/
 newtio.c_cc[VINTR] = 0; /* Ctrl-c */
 newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
 newtio.c_cc[VERASE] = 0; /* del */
 newtio.c_cc[VKILL] = 0; /* @ */
 newtio.c_cc[VEOF] = 4; /* Ctrl-d */
 newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
 newtio.c_cc[VMIN] = 1; /* blocking read until 1 character arrives */
 newtio.c_cc[VSWTC] = 0; /* '\0' */
 newtio.c_cc[VSTART] = 0; /* Ctrl-q */
 newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
 newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
 newtio.c_cc[VEOL] = 0; /* '\0' */
 newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
 newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
 newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
 newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
 newtio.c_cc[VEOL2] = 0; /* '\0' */

/*
   now clean the modem line and activate the settings for the port
*/
 tcflush(fd, TCIFLUSH);
 tcsetattr(fd,TCSANOW,&newtio);

}

void serial_end()
{
   tcsetattr(serialport,TCSANOW,&oldtio);
}
```

```
/*--------------------------*/
/* serial_close : */
/* */
/* ferme la liaison serie specifiee */
/* par serialport */
/*--------------------------*/
void serial_close()
{
        close(serialport);
 /* restore the old port settings */
 tcsetattr(serialport,TCSANOW,&oldtio);
}

/*--------------------------*/
/* serial_write : */
/* */
/* envoie la chaine string sur la liaison */
/* serie d'adresse serialport */
/*--------------------------*/
void serial_write_old(string)
char *string;
{
        tcflush(serialport,TCIOFLUSH); /* Vide le buffer */
        write(serialport,string,strlen(string));
        tcdrain(serialport); /* Attend la fin de lecture des caracteres */
}


/*--------------------------*/
/* serial_read : */
/* */
/* lit la chaine string */
/* sur la liaison serie d'adresse serialport */
/* renvoit la taille de la chaine ou 0 si erreur */
/* La chaine doit se terminer par un retour à la ligne */
/*--------------------------*/
int serial_read_simple(string)
char *string;
{
  int n=0;
  int taille=0;
  static char buff[300];

  buff[0]=string[0]=0;

  for(;;)
    {
      do
        {
          n=read(serialport,buff,300);
```

```
        }
      while(n<0);
      buff[n]=0;
      strcat(string,buff); /* Formation du mot */
      taille+=n;
      if (buff[n-1]=='\n') break;
    }
  return taille;
}




/*-------------------------------------*/
/* lecture sur la liaison série d'un message attendue */
/* On attend de recevoir un message commencant par c */
/*-------------------------------------*/

int serial_read(char *buff, char c)
 {
   int n,j,i;
   char buff2[500];
   n=strlen(buff);
   buff[n]=0x0D; buff[n+1]=0;
   i=0;
   do{
        tcflush(serialport,TCIOFLUSH);
        n=strlen(buff);
        write(serialport, buff ,n);
        tcdrain(serialport); /* attend que tout soit transmis */
        for(j=0;j<20000;j++)
        {
        n=read(serialport,buff2,100);
        if (n<0) { perror("Lecture buffer"); return; }
        if(n>0)
        {
         if(buff2[0]==c) return (1);
         else
           {
             printf("ERREUR lors de la lecture sur la RS232\n");
             printf("code retourne = %s \n",buff2);
           }
        }
      }
   i++;
   } while(i<8);
   printf("ERREUR lors de la lecture sur la RS232");
   printf("pas de ACK , temps depasse\n");
   exit(0);
 }

/* Fonction qui envoie le contenu du buffer et attend le retour du caractere c */
```

```
/* si tout est OK renvoie 1 sinon renvoie 0 */
/* 0 est notamment renvoye si la fonction recupere E sur la liaison serie */
/* E est le code pour une erreur cote micro controleur */

int serial_write_ack(char *buff,char c)
 {
   int n,j,i;
   char buff2[500];
   n=strlen(buff);
   /* buff[n]=0x0D; buff[n+1]=0;*/
   i=0;
   do{
        tcflush(serialport,TCIOFLUSH); /* detruit ce qui reste dans le buffer serie non lu et/ou non
envoye */
        n=strlen(buff);
        write(serialport, buff ,n);
        tcdrain(serialport); /* attend que tout soit transmis */
        for(j=0;j<20000;j++)
          {
            n=read(serialport,buff2,100);
            buff2[n]=(char) 0;
            if (n<0) { perror("Lecture buffer"); return 0; }
            if(n>0)
              {
                if(buff2[0]==c) return 1;
                else
                  {
                    if(buff2[0]=='E') return 0;
                    /*printf("ERREUR lors de la lecture sur la RS232\n");
                      printf("code retourne = %s \n",buff2); */
                  }
              }
          }
        i++;
   } while(i<8);
   printf("ERREUR lors de la lecture sur la RS232");
   printf("pas de ACK , temps depasse\n");
   exit(1);
 }
```