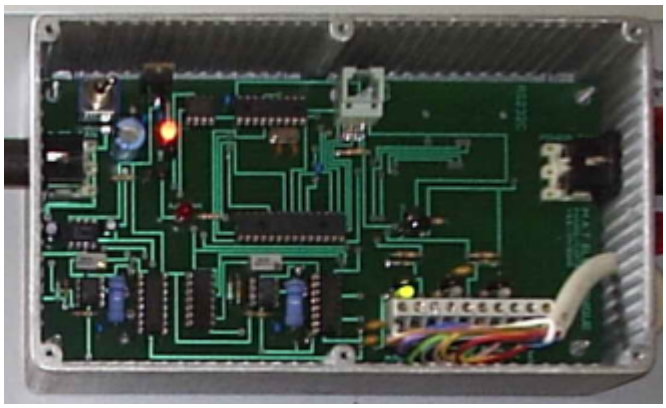
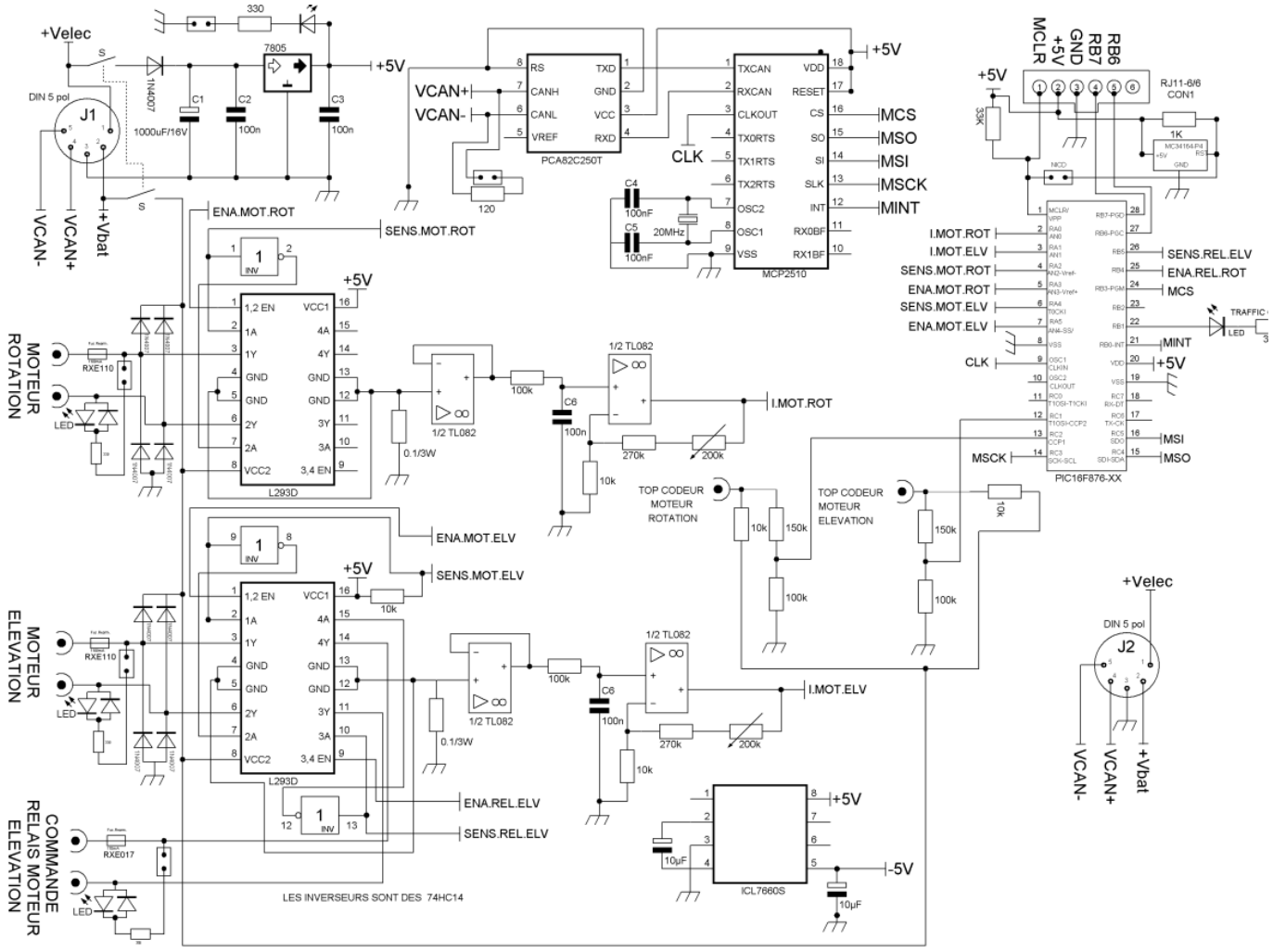
Présentation du module moteur.

Le module moteur permet la commande des moteurs de positionnement du panneau solaire.

Le module moteur dialogue avec le module de contrôle au travers du bus CAN.



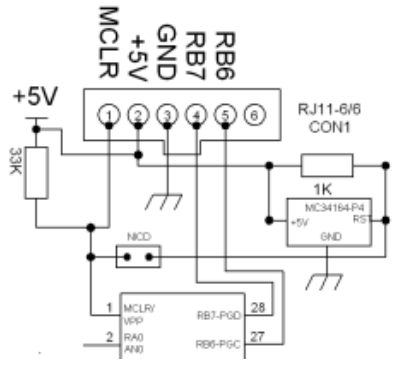
On donne le schéma structurel de ce module :



Gestion de la carte.

Le pilotage de la carte est confié à un microcontrôleur PIC16F876. La programmation in situ du PIC est assurée depuis le connecteur RJ11. Elle utilise les broches RB6 et RB7 du PIC. La broche MCLR permet, en mode programmation d'appliquer la tension de programmation.

L'horloge du processeur est fournie par la sortie SLK du circuit MCP2510 qui fournit un signal de fréquence 2,5 MHz (la fréquence du quartz divisée par 8).



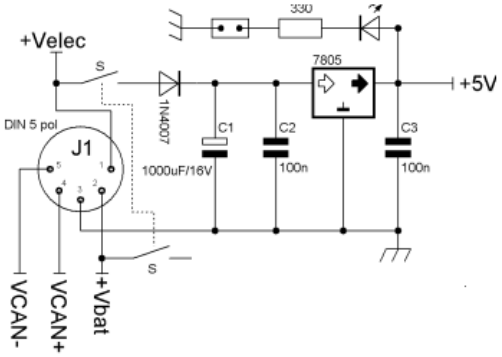
La broche MCLR permet également d'assurer un RESET matériel.

Le superviseur d'alimentation MC34164 est activé en mode normal par la mise en place d'un cavalier sur le connecteur NICD. Il permet de produire un RESET matériel dans le cas ou la tension d'alimentation devient trop faible.

Lors de la programmation ou la mise au point d'un programme, ce cavalier est à enlevé. Le microcontrôleur fonctionne à une fréquence de 4 MHz obtenue grâce à un quartz.

Alimentation de la carte.

L'alimentation de la carte est obtenue depuis le bus système.



La tension *+Velec* de 8V générée par le module énergie permet grâce au régulateur 7805 de fournir la tension de 5V nécessaire aux différents circuits de la carte.

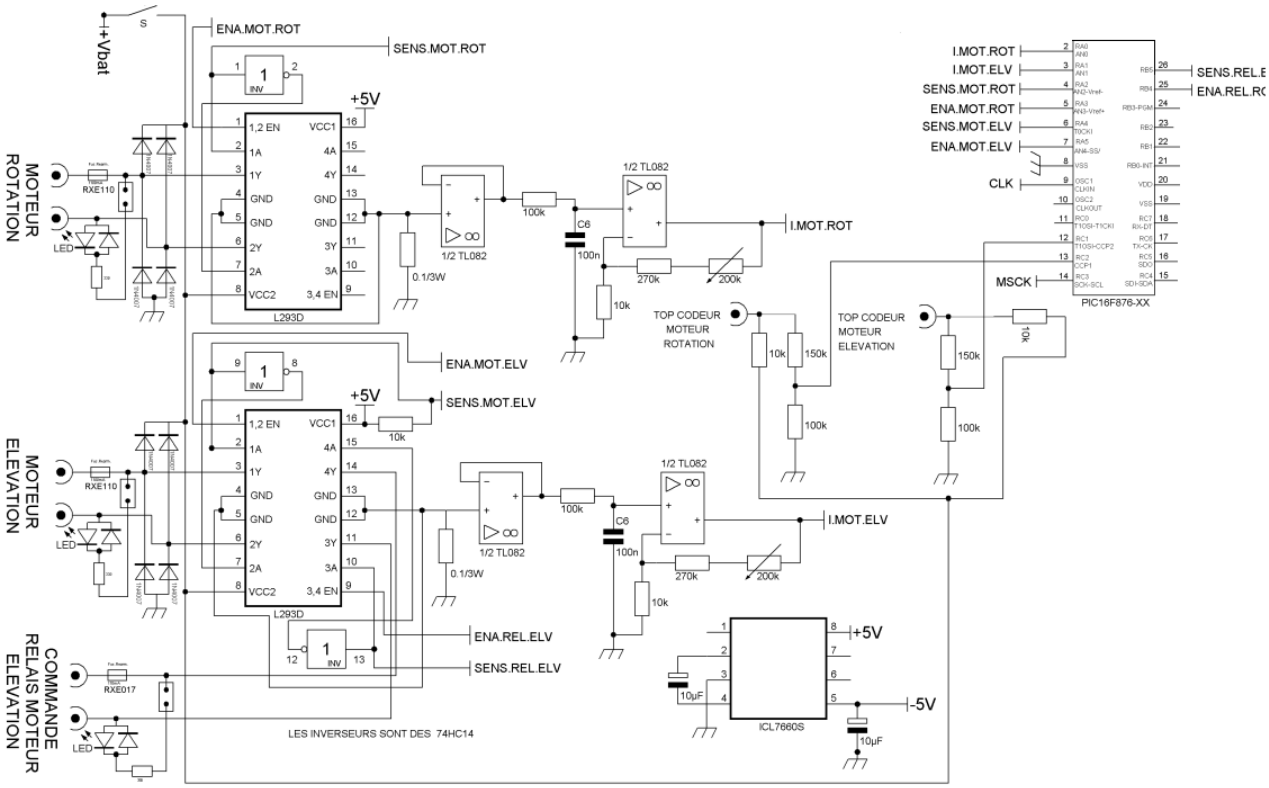
Les circuits de commande des moteurs sont alimentés par la tension *+Vbat* du bus système et générée par le module énergie.

La LED indicateur de présence tension peut être désactivée par le cavalier prévu à cet effet afin de diminuer la consommation.

Un interrupteur S permet de mettre la carte hors tension. Cette interrupteur peut être omis sur certaines versions (on a alors des « straps » et le module et continuellement sous tension).

Présentation du module moteur.

Commande des moteurs.



Le relais de commande, le moteur élévation et le moteur rotation sont commandés grâce aux circuits L293D. En sortie, on trouve les diodes de roue libre, une LED bicolore désactivable par un cavalier qui permet de visualiser les commandes et un fusible réarmable.

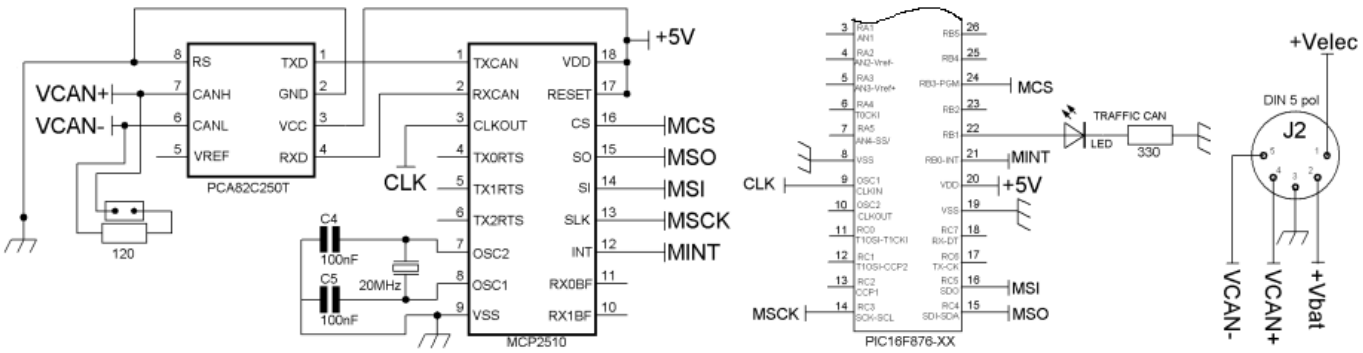
Les circuits L293 sont reliés à la masse à travers une résistance de 0,1 ohm. Les amplificateurs TL082 permettent de produire une tension proportionnelle au courant consommé. Les tensions sont appliquées aux entrées de conversion analogique/numérique du PIC afin de permettre la mesure du courant. Les amplificateurs sont alimentés en +5V et, grâce au convertisseur ICL7660S, en -5V.

Les codeurs solidaires du moteurs, dont la sortie est à collecteur ouvert (sur cette version mais ce n'est pas toujours le cas) fournissent des impulsions calibrées à 5V au PIC. Ces impulsions servent à déterminer la position du panneau.

Il est utile de se reporter au schéma de la carte moteur de la partie opérative pour analyser l'ensemble du schéma de commande des moteurs.

Accès au bus CAN.

Pour l'accès au bus CAN, on utilise les éléments suivants :



Le circuit MCP2510 dialogue avec le microprocesseur de la carte à travers les liaisons suivantes :

- Un bus série constitué de MSO, MSI et l'horloge MSCK.
- Un fils de sélection de boîtier MCS.
- La sortie MINT peut être utilisé pour générer des interruptions pour le microprocesseur (non activé dans notre cas).

La liaison du circuit MCP2510 avec le bus CAN s'effectue grâce aux broches TXCAN et RXCAN (niveau TTL) et le circuit d'interface PCA82C250T qui produit les signaux du bus. Le bus utilise les signaux VCAN+, VCAN- et GND.

Le circuit MCP2510 nécessite pour rythmer les signaux du bus CAN d'un quartz fixé ici à 20 MHz. La broche CLKOUT produit un signal de 2,5 MHz (division par 8 de la fréquence du quartz) qui est utilisé pour fournir une horloge au circuit PIC.

On notera la présence de la résistance RT de 120 ohms qui permet d'assurer, si nécessaire, la terminaison du bus CAN si l'on met en place le cavalier prévu.
La LED « TRAFFIC CAN » sera commandé par le PIC pour signaler une activité sur le bus CAN.

Etude du logiciel embarqué dans le PIC.

En ce qui concerne le logiciel, on trouve tout d'abord les indications relatives à la compilation du programme pour le PIC.

```
#include <16F876.h>
#device ICD=TRUE
#device *=16
#use delay(clock=2500000)
#fuses XT, NOPROTECT, BROWNOUT, NOWDT
#zero_ram                                   //remet la ram a 0 (initialise les variables a 0)
```

Gestion du bus CAN.

Le driver logiciel du circuit MCP2510 est pris en compte par la ligne :

```
#include "can.c"                                //driver can a consulter pour bits utilises
```

L'identifiant utilisé pour ce module est déclaré :

```
/* identifiants bus can */
#define i_m_m  0x400                  //reception d'une demande de donnee
#define i_mo   0x401                  //identifiant pour reponse au module de controle
#define i_m_c  0x402                  //reception d'un commande moteur
```

La commande de la LED d'activité est également déclarée :

```
#define led_on output_high(PIN_B1)       //led systeme
#define led_off output_low(PIN_B1)
```

Le bus CAN n'est utilisé qu'en réception par la fonction *gest_can* .

La réception est opérée par scrutation au moyen de la commande `if ( can_kbhit() )`

Si une réception est détectée, on examine l'identifiant et, suivant le cas, on recueille les données adéquates :

- La demande d'information depuis le module de contrôle correspond à l'identifiant *i_m_m*. La fonction *can_putd* permet de renvoyer les mesures demandées.
- L'envoie de données de commande correspond à l'identifiant *i_m_c*.

La fonction *gest_can* est appelée périodiquement par le programme principal.

```
void gest_can(){
   if ( can_kbhit() ){   //y a t il des donnees dans le buffer ?...
      if(can_getd(crx_id, &crxbuf[0], crx_len, rxstat)){ //...si oui lecture des donnees
         if (crx_id == i_m_m) {
            led_on;                                     //change la LED system
            ctxbuf[0]=mo_ie;
            ctxbuf[1]=mo_ia;
            ctxbuf[2]=mo_ovle;
            ctxbuf[3]=mo_ovla;
            ctxbuf[4]=on;
            ctxbuf[5]=ang_azim;
            ctxbuf[6]=ang_elev;
            ctxbuf[7]=butee;
            can_putd(i_mo,&ctxbuf[0],8,1,1,0);          //reponse avec 5 octets de buffer
            mo_ovle=0;                                  //raz indicateur de surcharge apres envoi
            mo_ovla=0;                                  //raz indicateur de surcharge apres envoi
         }
         if (crx_id == i_m_c) {
            led_on;                                     //change la LED system
            mot_a_sens=crxbuf[0];                       //sens moteur elevation
            mot_a_dure=crxbuf[1];                       //duree moteur elevation
            mot_e_sens=crxbuf[2];                       //sens moteur azimut
            mot_e_dure=crxbuf[3];                       //duree moteur azimut
            mot_off=crxbuf[4];
            park=crxbuf[5];
         }
      }
   }
   led_off;
}
```

Gestion des moteurs.

La gestion des moteurs est assurée par la fonction *gest_mot* appelée cycliquement depuis le programme principal.

On a deux cas :

1/ Si l'indicateur on est égal à 0, les moteurs sont à l'arrêt et il convient de les démarrer si la variable *mot_x_sens* vaut 1 ou 2 (0 = arrêt) ou si on a une demande de parking.

Pour chacun des déplacements demandés :

- On lance le moteur si la butée correspondante n'est pas activée.
- On fixe la durée du mouvement (500ms) multiplié par la donnée *mot_x_duree*.
- On positionne les bits d'entrée/sortie du PIC
- On attend 100 ms afin d'être sur que la prochaine mesure de courant trouvera bien un moteur actif.
- On positionne l'indicateur *on* qui nous renseigne sur le déplacement en cours.

Pour le mode *parking*, on lance le déplacement azimut puis le déplacement élévation.

```
void gest_mot(){
   if (on==0){
      if ((mot_e_sens==1)&&(but_h==0)){  //elevation montee
         mse=mot_e_dure*500;
         elev_rel_sens_high;
         elev_rel_enable_high;
         delay_ms(100);
         but_b=0;
         on=1;
      }
      if ((mot_e_sens==2)&&(but_b==0)){  //elevation descente
         mse=mot_e_dure*500;
         elev_rel_sens_low;
         elev_rel_enable_high;
         delay_ms(100);
         but_h=0;
         on=2;
      }
      if ((mot_a_sens==1)&&(but_d==0)){  //azimut sens horaire
         msa=mot_a_dure*500;
         azim_sens_high;
         azim_enable_high;
         delay_ms(100);
         but_g=0;
         on=3;
      }
      if ((mot_a_sens==2)&&(but_g==0)){  //azimut sens anti horair
         msa=mot_a_dure*500;
         azim_sens_low;
         azim_enable_high;
         delay_ms(100);
         but_d=0;
         on=4;
      }
      if (park==1){                      //parking
         if (but_g==0){
            azim_sens_low;
            azim_enable_high;
            on=5;
         }
         else if (but_b==0){
            elev_rel_sens_low;
            elev_rel_enable_high;
            on=6;
         }
         else if ((but_b==1)&&(but_g==1)){
            pos_azim=0;
            pos_elev=0;
            ang_elev=0;
            ang_azim=0;
            park=0;
         }
      }
```

2/ Si l'indicateur on est différent de 0, les moteurs fonctionnent et il convient de les arrêter si l'indicateur *off_flag* ou l'indicateur *mot_off* sont positionnés à 1. On positionne, lors d'une demande d'arrêt, les bits d'entrée/sortie à l'état inactif, on annule les demandes de mise en marche, on remet à 0 le compteur *mst* et les différents indicateurs.

```
if (on != 0){
   if ((off_flag == 1)||(mot_off==1)){ //on arrete les moteurs ?
      azim_enable_low;
      elev_rel_enable_low;
      mot_e_sens=0;                    //on annule les ordres de mise en marche
      mot_a_sens=0;
      mst=0;
      on=0;                            //pas de deplacement
      off_flag=0;
   }
}
butee=but_h+(but_b*2)+(but_g*4)+(but_d*8);          //calcul de l'indicateur butee
```

L'indicateur *butee* est calculé systématiquement en position les 4 bits de poids faible de l'octet.

Mesure des courants.

La mesure des courants est assurée par la fonction *mes_mot* appelée cycliquement depuis le programme principal.

Cette fonction est activée périodiquement par le *timer* logiciel grâce à l'indicateur *mes_flag*.

On calcule les angles élévation et azimut depuis les compteurs d'impulsions positionnés par les codeurs.

Les grandeurs sont codées sur 8 bits, aussi quand la rotation dépasse 255°, on ajoute 100 à l'ange d'élévation qui lui est toujours inférieur à 90°. Il suffira de tester les valeurs pour afficher correctement les angles au niveau du module de contrôle.

Pour l'élévation si la grandeur mesurée dépasse 240, on commande l'arrêt. Si le moteur lève le panneau, on positionne l'indicateur butée haute *but_h*.

```
void mes_mot(){
   int16 a;

   if (mes_flag==1){
      ang_azim=pos_azim/77;
      ang_elev=pos_elev/166;
      if (pos_azim > 19635){
         ang_elev = ang_elev + 100;
      }

      set_adc_channel(1);
      delay_us(10);
      a=read_adc();
      mo_ie = a/2;
      if (a>240){
         off_flag=1;                   //arret si surcharge
         mo_ovle=1;
         if(on==1){but_h=1;}           //si montee alors butee haute
      }
      if ((a<45)&&((on==2)||(on==6))){
         but_b=1;                      //alors butee basse
         off_flag=1;                   //si descente et plus de courant
      }
      set_adc_channel(0);
      delay_us(10);
      a=read_adc();
      mo_ia = a/2;
      if (a>180){
         off_flag=1;                   //arret si surcharge
         mo_ovla=1;
         if((on==3)||(on==5)){
            but_g=1;
            but_d=0;
         }                             //si gauche alors butee gauche
         if(on==4){
            but_d=1;
            but_g=0;
         }                             //si droite alors butee droite
      }
      mes_flag=0;
   }
}
```

Toujours pour l'élévation, si le courant s'annule alors qu'on descend le panneau (descente ou parking), c'est que le capteur de fin de course basse au niveau de la carte moteur de la partie opérative a coupé le courant. On annule la commande de déplacement et on positionne la butée basse.
Pour un déplacement en azimut, si la grandeur mesurée dépasse 180, on annule la commande de déplacement et on positionne les indicateurs de butée suivant le mouvement en cours.
Les indicateurs *mo_ovle* et *mo_ovla* sont positionnés si nécessaire et transmis au module de contrôle.

Le programme principal.

La déclaration des variables :

```
/* variables du systeme */
int16 mo_ie;                        //courant moteur 1
int16 mo_ia;                        //courant moteur 2
int mo_ovle;                        //indicateur de surcharge elevation
int mo_ovla;                        //indicateur de surcharge azimut
int mot_e_sens;                     //sens de deplacement 0,1 ou 2
int mot_e_dure;                     //duree de deplacement en secondes
int mot_a_sens;                     //sens de deplacement 0,1 ou 2
int mot_a_dure;                     //duree de deplacement en secondes
int on;                             //indique un eventuel deplacement
int park;                           //a 1 si demande parking depuis module de controle
int mot_off;                        //a 1 si demande arret depuis module de controle
int1 but_b;                         //a 1 si butee haute
int1 but_h;                         //a 1 si butee basse
int1 but_d;                         //a 1 si butee droite
int1 but_g;                         //a 1 si butee gauche
int ang_elev;                       //angle elevation
int ang_azim;                       //angle azimut
int butee;                          //etat des butees

long pos_elev,pos_azim;             //comptage des tops des codeurs
```

On déclare les fonctions :

```
/* declaration des fonctions */
void init();
void mes_mot();
void gest_can();
void gest_mot();
```

Reste le programme principal qui appelle le programme d'initialisation :

```
/* programme principal */
void main()
{
   init();
   while (1)
   {
      mes_mot();                    //mesure des courants
      gest_mot();                   //gestion des moteurs
      gest_can();                   //gestion bus can
   }
}
```

Le programme d'initialisation initialise le bus CAN, les variables, la conversion analogique/numérique, le *timer* et les entrées *CCPPx* de capture des impulsions des codeurs.

```
void init(){
   can_init();
   elev_sens_low;
   elev_enable_high;
   elev_rel_enable_low;
   azim_enable_low;
   mst=0;
   msc=0;
   on=0;
   park=0;
   mo_ovle=0;
   mo_ovla=0;
   but_b=0;
   but_h=0;
   but_g=0;
   but_d=0;
   off_flag=0;
   mot_e_sens=0;                //on annule les ordres de mise en marche
   mot_a_sens=0;
   mst=0;
   mse=500;
   msa=500;
   setup_adc_ports(RA0_RA1_RA3_ANALOG);
   setup_adc(ADC_CLOCK_INTERNAL);
   setup_ccp1(CCP_CAPTURE_RE);    // Configure CCP1 to capture rise
   setup_ccp2(CCP_CAPTURE_RE);    // Configure CCP2 to capture fall
   enable_interrupts(INT_CCP2);   // Setup interrupt on falling edge
   enable_interrupts(INT_CCP1);   // Setup interrupt on falling edge
   setup_timer_2(T2_DIV_BY_16,8,5);    // parametrage du timer 2
   setup_timer_1(T1_INTERNAL);    // Start timer 1
   enable_interrupts(INT_TIMER2);      // autorisation interruption timer2
   enable_interrupts(GLOBAL);          // autorisation de toutes les interruptions (afin de prendre en compte timer2)
}
```

Présentation du module moteur.

Gestion du temps.

Afin de ne pas rendre les fonctions bloquantes, on utilise un *timer* logiciel appelé par un *timer* matériel initialisé pour générer une interruption toutes les ms.

Les déclarations :

```
/* indicateurs des taches activees periodiquement */
int1 off_flag;                          //indicateur de mise a l'arret
int1 mes_flag;                          //indicateur de declenchement mesure i
int16 msa;                              //tick de comptage azimut
int16 mse;                              //tick de comptage elevation
int16 mst;                              //tick de comptage du time out moteur
int16 msc;                              //tick de comptage pour acquisition courant
```

Toutes les millisecondes, la fonction ci-dessous est déclenchée :

```
/* timer des taches appele par interruption */
#int_timer2
void isr_timer2(void) {
   msc++;
   if (msc == 100){
      mes_flag = 1;
      msc=0;
      }
   if ((on != 0)&&(park==0)){mst++;}                //timer qui est appele toutes les ms par une interruption du timer2
   if ((mst >= mse)&&(park==0)){
      off_flag = 1;
      mst=0;
      }       //on incremente mst si moteur on, qd mst>=mse on positionne flag pour arret
   if ((mst >= msa)&&(park==0)){
      off_flag = 1;
      mst=0;
      }       //on incremente mst si moteur on, qd mst>=msa on positionne flag pour arret
}
```

L'indicateur *msc* sert à déclencher les mesures de courants. Il est activé toutes les 100 ms.

Si on a un mouvement (*on* non nul), on incrémente le compteur *mst* sauf en mode *parking* qui s'arrête quand in arrive en butée.

Dés que *mst* atteint la valeur *msa* (durée déplacement azimut) ou *mse* (durée déplacement élévation), on arrête le déplacement (*off_flag* = 1) et on remet *mst* à 0.

Gestion des codeurs.

La gestion des codeurs est activée par 2 interruptions qui suivant le sens et la nature du déplacement vont incrémenter les compteurs d'impulsions.

```
#int_ccp1
void isr_cpp1() {
   if (on==3){
      pos_azim=pos_azim+1;
   }
   else if (((on==4)||(on==6))&&(pos_azim>>0)){
      pos_azim=pos_azim-1;
   }
}

#int_ccp2
void isr_cpp2() {
   if (on==1){
      pos_elev=pos_elev+1;
   }
   else if (((on==2)||(on==5))&&(pos_elev>>0)){
      pos_elev=pos_elev-1;
   }
}
```

Les compteurs *pos_elev* et *pos_azim* sont remis à 0 quand on arrive en butée car un parking est systématiquement activé en fin de journée, à la mise sous tension ou manuellement par l'utilisateur.