Un apercu du code de la carte principale de Snooky

Florent de LAMOTTE

24 septembre 2002

Cette documentation a pour but de permettre au lecteur de mieux comprendre l'organisation du code source de Snooky.

Table des matières

1 Organisation du code

Le code de la carte principale de Snooky est, à l'exception du code de démarrage *crt0.s*, entièrement écrit en C.

Ce code est localisé dans un seul répertoire contenant à la fois le code source, le Makefile permettant d'obtenir des binaires, le script pour l'éditeur de lien ... en fait tout ce qui concerne le développement pour la carte.

Le code à été décomposé en modules pour apporter une meilleure structuration. Pour chaque module, on trouve un fichier source écrit en C, ainsi que son header qui contient des déclarations de types, les variables globales ainsi que les prototypes des fonctions qui peuvent être appellées à l'exterieur de ces modules.

Le fichier *Changelog* présent dans le répertoire du code contient un récapitulatif des changements effectués sur ce même code. Le fichier *TODO* contenait lui les différentes choses qu'il restait à implémenter. Ces deux fichiers n'ont pas forcément été très bien mis à jour[1] mais reflètent les grandes modifications qui ont été apportées au code.

Des archives de ce code ont été faites de manière régulière et on été timestampées, il devrait être assez aisé de retrouver le code tel qu'il était pendant les différentes phases de conception de la carte. Les dates d'archivage correspondent normalement à l'écriture d'une nouvelle entrée dans le *Changelog*.

[1] surtout en ce qui concerne le fichier TODO et pour les modifications de dernière minute

2 Les fonctions de bas niveau

Trois modules s'occupent de fonctions de bas niveau :
- *timer.c* contient l'interruption timer et la routine *sleep* permettant de forcer une attente d'un certain nombre de secondes.
- *serial.c* contient une interruption permettant de gérer la liaison série. Cette interruption est fonctionnelle et a été debuggée mais n'a pas été utilisée.
- *pic_i2c.c* s'occupe de la gestion du PIC pour la communication sur le bus I2C, ce module a été détaillé dans le document concernant la communication entre le PIC et le 332 [2].

2.1 L'interruption timer

Le code de l'interruption timer se situe dans le module *timer.c*.
L'action principale effectuée lors de cette interruption concerne le polling sur le bus I2c :

```
test_pic
leds ^= LED_ROUGE;
if (get_status()==0xFF)
leds &= ~LED_VERTE1;
else
leds |= LED_VERTE1;
if (get_meca_status()==0xFF)
leds &= ~LED_VERTE2;
else
leds |= LED_VERTE2;

get_meca_couleur_boule();

release_pic
```

Comme on peut le voir dans ce bout de code tiré de l'interruption, on récupère le status de la carte d'asservissement et de la carte méca. Suivant l'état de chacune des cartes, on positionne correctement les leds. Les fonctions *get_status* et *get_meca_status* mettent aussi à jour des variables concernant la carte mécanique et la carte électronique, utilisées dans les fonctions du programme.

L'appel à *get_meca_couleur_boule* permet de récupérer la couleur des boules présentes dans le barillet et mets donc à jour les variables correspondant pour des calculs de stratégie.

L'appel aux macros test_pic[2] et releases_pic est obligatoire pour s'assurer que l'on accède pas au pic dans le code alors que l'on y accède dans une interruption et vice-versa.

Cette partie du code est aussi responsable du clignotement de la led rouge qui indique que la liaison i2c est fonctionnelle[3].

On demande aussi dans l'interruption des nouvelles de notre ennemi, principalement pour des raisons stratégiques :

```
test_pic
```

[2]Dans l'interruption on utilise test_pic qui n'est pas blocante pour justement ne pas bloquer le processeur alors que dans le code principale, on utilise get_pic qui lui est bloquant

[3]La signification réelle du clignotement de cette led est le code concernant l'i2c dans l'interruption timer est executé, ce qui signifie effectivement que l'i2c n'est pas bloqué quelque part dans la boucle principale.

```
get_info_adv();
release_pic

if (adv_nearby_panier)
        paniers_visites_adv[adv_panier] = 1;
```

La fonction *get_info_adv* s'occupe en effet de demander des informations concernant l'ennemi à la carte d'asservissement et de mettre à jour les variables correspondantes.

Ici, si l'adversaire est proche d'un panier, on positionne un flag indiquant qu'il est allé le visiter. Ce code permet de savoir quels paniers on peut aller piller.

La deuxième fonction de ce code d'interruption concerne le debugging.

Tout d'abord, en ce qui concerne les cartes sur le bus I2C, on positionne les leds selon l'état (connecté ou déconnecté) de chaque carte, les leds correspondantes sont les deux leds vertes.

Ensuite, on utilise la led jaune pour refléter l'état actuel de la connection avec la caméra.

```
switch (cam_error) {
case 0:
        leds |= LED_JAUNE;
        break;
case 100:
        if (system_time % 2 == 0)
                leds ^= LED_JAUNE;
        break;
case 101:
        leds &= ~LED_JAUNE;
default:
        leds &= ~LED_JAUNE;
        break;
}
```

Le code suivant n'est effectif que si le dernier jumper est relevé. Il permet d'envoyer dans ce cas, au travers de la liaison série, des informations au programme *simu_cam* [1].

```
if ((system_time % 2 == 0) && ((SIM.PORTE0 & 0x10))) {
        test_pic
        get_position();
        get_position_adv();

        get_meca_mode_fonctionnement();
        release_pic

        send_position();
        send_asser_status();
        send_meca_status();
        send_info_adv();
}
```

La dernière partie de cette interruption permet à l'utilisateur du robot de savoir si le robot est prêt à recevoir la tirette. Dans ce cas, les leds perdent leur signification normalent et changent d'etat toutes ensemble toutes les secondes.

```
        if (ready) {
                LEDS = leds;
        } else {
                if (system_time % 4 == 0) {
                        LEDS = ~LEDS & (leds | LED_ROUGE);
                }
        }
```

2.2 L'initialisation de la carte principale

L'initialisation de la carte principale s'effectue dans la fonction *main*, localisée dans le module *snooky_brain*.

Les lignes qui suivent tentent de décortiquer la phase d'initialisation de la carte principale. j'avouerais que cette phase d'initialisation est un peu chaotique. Avec un peu d'explications, j'espère que ça deviendra un peu plus clair.

```
        /*
         * Initialisation du 332
         */
        empty_i2c_buffer();

        get_pic
```

On commence par intialiser les routines i2c et on réserve le pic. Celui-ci est réservé pendant toute la phase d'initialisation du programme.

```
        // Fsysteme = Fquartz[4*(60+1)*2]=15990784 Hz
        SIM_SetClock (0, 1, 55);
//      SIM_SetClock (0, 1, 60);
        SCI_Init ();
```

Les routines *SIM_SetClock* et *SCI_Init* fait partie des routines de la *libESEO*. On ajuste la vitesse du 68332 à 15Mhz pour pouvoir ensuite faire fonctionner la liaison série à 115200 bps.

La fonction *SCI_Init* fixe les paramètres par défaut du port série.

- 9600 bps
- 8 bits pas de parité
- interruption timer désactivée

```
        exception_table.auto_vector[4] = (longword) I2C_int;

        /* Configuration de cs5 pour repondre a l'int */
        SIM.CSBAR5 = 0xFFF8;
        SIM.CSOR5 = 0x2C0B;
        SIM.CSPAR0 |= 0x0800;

//      exception_table.user[2] = (longword) int_serial;
        exception_table.user[2] = (longword) int_cam;
        QSM_SetIntVector(66);
        SCI_SetIntLevel(6);
        //SCI_ReceiverInt(ENABLE);
        QSM.SCCR1Low.RIE = 1;
```

Il nous faut configurer les interruptions. Dans le fichier header *mc68332.h* nous avons défini une correspondante entre le tableau *auto_vector* et les interruptions autovectorisées.

Il faut faire attention, le premier vecteur auto vectorisé correspond à la première entrée dans le tableau. Ceci signifie que l'auto-vecteur 1 correspond à la variable exception_table.auto_vector[0].

Ici, on assigne donc à l'interruption *I2C_int* l'interruption 5.

Le code utilisé pour gérer l'interruption timer est *int_cam*, c'est une version améliorée de *int_serial* qui peut gérer directement une partie de la communication avec la caméra. Cette possibilité n'est par contre pas utilisée. (voir la partie concernant la caméra)

De plus pour autoriser l'interruption série, on écrit directement dans le registre, sans passer par les librairies (la fonction utilisant la librairie est composée). Ceci est un vieux relicat des longues heures de debugging.

```
/* pe7 en sortie (reset du PIC) */
SIM.DDRE |= 0x80;
SIM.PORTE0 &= 0x7F;


/* pf3467 en sortie */
SIM.PFPAR |= 0x20;
SIM.DDRF  |= 0xD8;

SIM.PORTF0 = 0; /* on eteinds les leds */
```

Ensuite on configure les ports d'entrée-sortie pour le carte externe.

```
/* Ajustement du masque d'interruption */
asm("move.w     #0x2200, %sr");

/* force la vitesse du port serie
 * 9 => 57400  (F = 16MHz)
 * 8 => 57400  (F = 14MHz)
 * 5 => 115200 (F = 14MHz)
 */
QSM.SCCR0 = 4; // 115200@14MHz

exception_table.user[0] = (longword) int_timer;

SIM_SetIntVector(64); /* 64 : 1er vecteur utilisateur */
SIM_SetIntLevel(4); // 4 pour l'enabler
SIM.PITR = 0x0102; // periode = PITM/16
```

On configure l'interruption timer.

```
/*
 * Initialisation du programme
 */

init_terrain();
I2C_reset();
```

```
cam_init();

release_pic
```

Finalement, on initialise le programme et on relache le pic.

3 La gestion de la carte d'asservissement

Les routines qui permettent de gérer la carte d'asservissement sont localisées dans le module *deplacement.c*.

Ces routines utilisent les routines de gestion du bus I2C dont les prototypes sont dans l'header *i2c.h*.

Les messages envoyés sur le bus sont composés de la commande, suivie des différents paramètres. Le tableau suivant présente ces différentes commandes, explicite les paramètres et donne le nom des fonctions utilisant ces commandes.

COMMANDE	**PARAMETRES**	**Description**	**Fonction**
GOTOXY	x (poids fort) x (poids faible) y (poids fort) y (poids faible) options (NA)	Demande à la carte d'asservissement de déplacer le robot jusqu'à une position souhaitée	goto_position
GOTO_REVERSE	deplacement	demande au robot d'effectuer un déplacement en marche arrière	goto_reverse
GOTO_PANIER	panier	Demande à la carte d'asservissement de déplacer le robot jusqu'au panier	goto_panier
ROTATION_RAW	angle (2)	Demande à la carte d'asservissement de faire effectuer au robot une rotation	rotate_robot
STOP	aucun	Arrête le robot	stop_robot
RESET_POS	aucun	Réinitialise la position du robot	reset_position
SET_VITESSE_TRAJ	vitesse acceleration	Configure la vitesse du robot en lignes droites	set_traj_params
SET_VITESE_ROT	vitesse acceleration	Configure la vitesse du robot en rotation	set_rot_params

On peut aussi récupérer des informations en provenance de cette même carte d'asservissement. Pour récupérer des informations, il faut positionner le type de retour de la carte d'asservissement à l'aide de la fonction *asser_set_type_r*. Cette fonction change le type de retour de la carte d'asservissement si celui-ci doit être changé. Elle est automatiquement appellée par les fonctions citées ci-après.

Les types de retour pour ces informations sont les suivants.

Type de retour	Données retournées	Description	fonction
POSITION	position_x (poids fort) position_x (poids faible) position_y (poids fort) position_y (poids faible) angle (poids fort) angle (poids faible)	Retourne la position actuelle du robot ainsi que son angle.	get_position
POSITION_BALISE	position_x (poids fort) position_x (poids faible) position_y (poids fort) position_y (poids faible) angle (poids fort) angle (poids faible)	Retourne la position actuelle du robot adverse ainsi que son angle.	get_position_adv
CHECK_BUSY	*status* PB_GRAVE BUSY_PANIER BUSY_GOTO_PANIER CHOC_PANIER(NA) CHOC_BORD(NA) PB_MOTEUR	Retourne l'état de la carte d'asservissement	get_status
INFO_ADV	*data_retour* panier(2 bits) GOING_PANIER NEARBY_PANIER GOING_ROBOT NEARBY_ROBOT	Retourne les informations sur le robot adverse	get_info_adv

3.1 Les variables conservant l'état de la carte d'asservissement

Plusieurs variables servent à conserver l'état de la carte d'asservissement. Elles sont mises à jours lors de l'appel à des fonctions permettant de récupérer des informations en provenance de la caméra.

Ces variables sont toutes des variables globales et il aurait sûrement été préférable de placer ces informations dans des structures ...

3.1.1 La position des robot

La position du robot est conservée dans les variables suivantes :
- *position_x* conserve la coordonnée x de la position du robot
- *position_y* conserve la coordonnée y de la position du robot
- *angle* conserve l'angle du robot
- *position_rayon_err* (N/A)

Ces variables représentent la position du robot lorsque la dernière fonction *get_position* a été appellée.
On conserve aussi la position du robot adverse de la même manière :
- *position_adv_x* conserve la coordonnée x de la position du robot
- *position_adv_y* conserve la coordonnée y de la position du robot
- *angle_adv* conserve l'angle du robot
- *position_adv_rayon_err* (N/A)

On conserve aussi les coordonnées du point vers lequel on essaie de se diriger, et cela pour les reprises d'erreurs.

– *position_desiree_x*
– *position_desiree_y*

3.1.2 Les paramètres de la trajectoire

Les paramètres de la trajectoire sont conservés dans les variables suivantes :

– *position_vitesse_traj*
– *position_vitesse_rot*
– *position_accel_traj*
– *position_accel_rot*

Ces paramètres sont mis à jour lors des appels aux fonctions *set_traj_params* et *set_rot_params*.

3.1.3 Status de la carte d'asservissement

Le status de la carte d'asservissement est conservé dans une variable *asser_status* qui elle même est décomposées en plusieurs variables booléennes représentant l'état de la carte d'asservissement.

– *asser_pb_grave*
– *asser_busy_trajet*
– *asser_busy_goto_panier*
– *asser_choc_panier*
– *asser_choc_bord* (N/A)
– *asser_pb_moteur*

Ces variables sont toutes mises à jour lors des appels à *get_status*.

3.1.4 Informations sur l'adversaire

Les informations concernant l'adversaires sont stockées dans la varible *info_adv* et dispatchées dans les variables booléennes suivantes :

– *adv_going_panier* L'adversaire se dirige vers un panier
– *adv_nearby_panier* L'adversaire se situe à proximité d'un panier (rayon)
– *adv_going_robot* L'adversaire se dirige vers nous
– *adv_nearby_robot* L'adversaire est à proximité

Ces informations sont calculées par la carte d'asservissement et sont utilisées pour éviter l'adversaire et pour piller les paniers. Elles sont récupérées lors de l'appel à *get_info_adv*

3.1.5 Ralentissement et réaccélération

La variable *reprise* est passée à 1 par la fonction *ralentir_roobot*, utilisée lorsque le robot est à proximité d'un adversaire. Elle permet à la fonction *reaccelerer_robot* de savoir si effectivement le robot est ralenti.

Pour réaccélérer le robot, on restitue les paramètres de la trajectoire tels qu'ils étaient lors de l'appel à *ralentir_robot*. C'est pour cela que ces paramètres sont enregistrés dans les variables *vitesse_normale* et *acceleration_normale*.

4 La gestion de la carte meca

Les ordres que l'on envoie à la caméra sont toujours constitués de deux octets.
Voici les deux commandes que l'on peut envoyer à la carte méca :

COMMANDE	**PARAMETRES**	**Description**	**Fonction**
CHANGER_MODE	nouveau mode	Permet de changer le mode dans lequel est la carte méca.	set_meca_mode_fonctionnement
ENVOI_ORDRE	ordre	Envoie un ordre à la carte méca. Ces ordres ne sont effectifs que si la carte méca est en mode panier.	set_meca_ordre

Voici les différents états de la carte méca.

Etat	**Description**
RAMASSAGE	Mode dans lequel le robot ramasse les boules sur le terrain. Les bras sont ouverts et les tapis sont en fonctionnement.
COQUILLE	Dans ce mode les deux bras sont fermés et les moteurs sont arrêtés. C'est un mode de protection dans lequelles les balles ne peuvent pas entrer dans le robot.
PANIER	C'est le mode dans lequel est le robot quand il décharge les boules. Dans ce mode, la carte méca peut recevoir des ordres, sinon ces ordres sont rejetés.

Et les différentes commandes qui peuvent êtres reçues par le robot quand il est en mode panier.

Commande	**Description**
RAMASSER_BOULE	Lorsque le robot est en mode panier, fais fonctionner l'avalement pour avaler une boule sous un panier.
DECHARGER_BR	Décharge une boule rouge.
DECHARGER_BN	Décharge une boule noire.

Il est aussi possible de récupérer l'état acutel de la carte méca. Voici les différentes informations qui peuvent être récupérées.

Type de retour	**Données retournées**	**Description**	**fonction**
CHECK_BUSY	*status* PB_GRAVE BUSY_BARILLET RAMASSE_TERRAIN RAMASSE_PANIER DECHARGE_PANIER BUSY_TAPIS BARILLET_PLEIN	Retourne le status de la caméra.	get_meca_status
ETAT	boules rouges (4 bits faibles) boules noires (4 bits forts)	Renvoie l'état du barillet (nombre de boules de chaque couleur).	get_meca_couleur_boule
MODE	mode	Renvoie le mode actuel du robot.	get_meca_mode

4.1 Les variables conservant l'état de la carte méca

Le mode actuel de la caméra est enregistré dans la variable *meca_mode*
Des variables contiennent le nombre de boules :

Variable	Signification
meca_nb_boules	Nombre total de boules dans le barillet
meca_nb_boules_noires	Nombre de boules noires dans le barillet
meca_nb_boules_rouges	Nombre de boules rouges dans le barillet

Le status de la carte méca est enregistré dans la variable *meca_status*. Cette variable est décortiquée dans les variables suivantes :

Variable	Signification
meca_pb_grave	N/A
meca_busy_barillet	Le barillet est en rotation
meca_ramasse_terrain	Le robot est en train de ramasser une boule prise sur le terrain
meca_ramasse_panier	Le robot est en train de ramasser une boule prise sous le panier
meca_busy_tapis	Le tapis est en foncitonnement
meca_barillet_plein	Le barillet est plein

5 La gestion de la camera

La carte de vision, gérant les deux caméras n'est pas sur le bus I2C, contrairement aux autres cartes. Elle possède sa liaison privilégiée avec la carte principale, au travers d'une liaison série.

Les fonctions gérant la communication avec la carte de vision sont placées dans le module *camera.c*.

5.1 Initialisation de la carte de vision

L'initialisation de la carte de vision se fait dans la fonction *cam_init*.

Cette fonction envoie six **NACK (0xAA)** ce qui normalement fait sortir la carte de vision de tous les niveaux de sa machine d'état et donc la réinitialise.

Si la carte de vision ne réponds pas dans un temps correct, elle est considérée comme étant non fonctionnelle ou tout simplement pas là. La variable *cam_dead* est alors positionnée à 1 pour éviter toute communication avec la camera qui n'est de toute manière pas là.

5.2 Fonctionnement de la carte de vision

Lorsque l'on désire communiquer avec la carte de vision, il faut le lui signaler, à l'aide de la fonction *cam_init_com*. Cette fonction envoie un caractère **ACK (0x55)** à travers la liaison série et attends un **ACK** en provenance de la carte de vision. Si un signal **NACK (0xAA)** est reçu, cela signifie qu'il y a eu une erreur et on la traite.

La caméra possède trois modes de fonctionnement. Le changement de mode s'effectue à l'aide de la commande **CAM_MODE** suivie du mode désiré.

Les trois modes possibles pour la carte de vision sont :

- Un mode *CAM_SCAN* dans lequel la caméra pivote sur elle même pour parcourir tout le terrain et récupérer la position de toutes les balles. Ce mode est utilisé au démarrage du robot.
- Un mode *CAM_SHOT* dans lequel la caméra prends une seule photo à chaque fois qu'elle en recoit l'ordre
- Un mode *CAM_TRACK* non implémenté qui aurait du permettre de suivre une balle

Mode	Valeur
CAM_SCAN	10
CAM_TRACK	20
CAM_SHOT	30
CAM_SCAN_START	40

TAB. 1 – Les modes de la caméra

Lorsque l'on veut prendre une photo, on envoie la commande **CAM_GRAB**, suivi du numéro de la caméra et de la position du robot (position_x(poids fort) position_x(poids faible) position_y(poids fort) position_y(poids faible) angle(poids fort) angle(poids faible).). La fonction gérant cette commande est la fonction *cam_grab*. Il existe en fait une commande **CAM_GRAB1** et une commande **CAM_GRAB2** permettant de grabber à l'aide de la caméra avant ou à l'aide de la caméra arrière, mais la fonction *cam_grab* prends en paramètre un entier correspondant à la caméra que l'on souhaite utiliser et envoie la bonne commande à la carte de vision.

On récupère l'information sur la position des boules à l'aide de la commande **CAM_INFOS**. Cette commande provoque l'envoi par la carte de vision de la position des balles qu'elle a repérées lors du dernier ordre qu'on lui a envoyé. La fonction gérant cette récupération des informations au niveau de la carte principale est la fonction *cam_get_infos*.

Commande	Valeur
CAM_MODE	10
CAM_INFOS	20
CAM_STATUS	30
CAM_GRAB1	40
CAM_GRAB2	50

TAB. 2 – Commandes supportées par la caméra

On termine la communication en apellant la fonction *cam_end_com* qui envoie un caractère **NACK** à la carte de vision.

5.3 Le mode scan

Le mode scan, est lui-même dédoublé en deux cas :

- Le mode **CAM_SCAN** utilisé pour scanner le terrain en cours de match.
- Le mode **CAM_SCAN_START** utilisé pour scanner le terrain au démarrage. Dans ce mode, la carte de vision ne prends en compte que les balles qui sont sur la première moitié du terrain et les place aux intersections des lignes blanches. Elle détermine la position des autres balles à l'aide de la symétrie.

Une fois que la carte de vision est placée dans le mode **CAM_SCAN**, il faut lui envoyer la commande **CAM_GRAB** pour lancer effectivement la commande de scan sur la caméra souhaitée.

A ce moment la caméra désignée va commencer à tourner sur elle même pour prendre un maximum de photos de la disposition des boules sur le terrain.

Lorsque l'on veut récupérer les informations, on envoie un ordre **CAM_INFOS** et on récupère la position des boules envoyée par la carte de vision.

Pour que la caméra s'arrête de tourner, il faut lui faire changer de mode.

5.3.1 Le mode shot

Le mode shot permet de prendre une seule photo et de récupérer ainsi la position des boules placées devant une caméra.

Pour utiliser ce mode, il faut passer la caméra en mode **CAM_SHOT**, lancer une commande de capture à l'aide d'un **CAM_GRAB** et récupérer l'information à l'aide d'un **CAM_INFOS**.

Lorsque la carte de vision est en mode shot, elle ajuste l'angle de vision de la caméra pour qu'elle ne regarde jamais à l'extérieur du terrain.

5.3.2 L'attente par it de la carte de vision

int_cam est une nouvelle version de *int_serial*, qui s'occupe de la communication avec la carte de vision lorsque l'on utilise la fonction *cam_init_com_it* pour initialiser cette même communication.

Ce mode de fonctionnement permet de lancer une initialisation de la communication avec la carte de vision et d'aller s'occuper d'autre chose en attendant que celle-ci soit prête (le reste de la communication se déroulant normalement).

Ce mode n'a néamoins pas été utilisé vu que la carte de vision répondait dans des temps raisonnables.

5.3.3 Les variables conservant l'état de la connection avec la carte de vision

La variable *cam_error* contient le dernier code d'erreur en provenance de la carte de vision. Ce code est répercuté sur les leds d'état.

La variable *cam_dead* est positionnée à 1 lorsqu'il n'y a plus aucun espoir d'établir la connection avec la carte de vision. Dans la réalité, cela arrive quand on n'arrive pas a communiquer avec la carte de vision pendant la routine d'initialisation de celle-ci : *cam_init*.

cam_ready est utilisée dans la gestion de la caméra par interruption pour indiquer que la caméra à répondu au **ACK** par un **ACK**.

nb_balles_c est un tableau contenant le nombre de balles de chaque couleur, aperçues par la caméra.

Finalement *cam_state* est utilisée dans la gestion par interruption pour indiquer l'état de la caméra.

6 Gestion du terrain

7 La machine d'état

La machine d'état gère le comportement global du robot. Et comme celui-ci possède un grand nombre de fonctionnalités, celle-ci est plutôt complexe.

La décomposition de cette machine d'état à été conçue pour simplifier au maximum toute la gestion du robot. Au final on obtient quelque chose de souple et modulaire.

7.1 Principe

Le principe est de décomposer le fonctionnement du robot en quelques super-etats. Ces états définissent en quelque sorte le but du robot comme par exemple *sortir de l'aire de départ*, *ramasser les balles sur le terrain*, *décharger le barillet* ...

Chacun de ces états est ensuite décomposé lui-même en sous-états, définissant la séquence d'actions à effectuer.

[Un schema c'est ce qu'il y a de mieux pour expliquer]

7.2 Fonctionnement de la machine d'état

7.2.1 Les super-états

Les super-états sont composés chacun trois trois fonctions :

- Une fonction exécutée lorsque l'on entre dans l'état, chargée de tout initialiser : *init*
- Une fonction qui sera appellée periodiquement par la boucle principale, qui fera évoluer la sous machine d'états : *main*
- Une fonction chargée de tout nettoyer lors de la sortie de l'état : *close*

7.2.2 Le déroulement de la machine d'état

Le module *state_machine.c* contient le code permettant de gérer la machine d'état.

La fonction *change_state* permet de changer d'état.

La fonction *find_state* n'a pas été implémentée. Son but aurait été de définir dans quel état il serait plus intéressant d'aller après, compte tenu de l'état duquel on vient et de l'état actuel du robot (nombre de balles, position, position de l'adversaire, etc ...). Cette fonction n'ayant pas été implémentée, le choix de l'état dans lequel on va se fait toujours dans l'état duquel on vient (ce qui devient lourd à la longue). Cette fonction *find_state* aurait de plus permis d'ajouter un peu d'aléatoire dans le fonctionnement de la machine d'état.

La fonction *states_main* correspond à la boucle principale de la machine d'état. C'est ici qu'on s'occupe de l'évitement de l'adversaire et de déclencher une séquence particulière au bout d'une minute de matche. Cette séquence consiste à ramasser les balles autour du terrain et de vider le barillet. Elle sert aussi de watchdog, reveillant le robot en cas d'endormissement (ca a été utile en matche :)). Enfin, on appelle la fonction *main* de l'état en cours, ce qui permet de faire tourner la machine d'état.

L'état actuel est stocké dans un pointeur vers une variable de type *State*, appellée *current_state*. L'état précédent est lui enregistré dans la variable *prec_state*.

Le type *State*, défini dans *state_machine.h* définit une structure dans laquelle on trouve des pointeurs sur les fonctions *init*, *close* et *main* de l'état.

```
typedef struct {
        state_id id;
        int (* init) (void);
        int (* close) (void);
```

```
        int (* main) (void);
} State;
```

7.2.3 Implémentation des super-états

Chaque super-état est implémenté dans un fichier différent, contenant une structure de type *State*[4], définissant cet état, les trois fonctions décrivant l'état : *init*, *close* et *main* et une énumération de tous les sous-états.

7.2.4 Les sous machines d'état

La fonction *main* de notre super-état contient le switch de la sous machine d'état, définissant par là même les sous-états et l'enchaînement entre ceux-ci.

7.3 La machine d'état

Nous en venons maintenant au fonctionnement de la machine d'état. Celle-ci est composée des super-états suivants :

- arrete
- depart
- ramassage
- dechargement
- pillage
- defense
- gene
- victoire
- defaite
- ramassage_aveugle
- ramassage_autour

Dans chacun des super-états est développée une sous machine d'état.

[4]Les structure définissant les états sont toutes déclarées dans *state_machine.h* (mais définies dans le fichier .c correspondant à l'état).

7.3.1 Description des super-états

Etat	Description
arrete	Le robot reste arrêté. Cet état permet en fait d'arrêter le robot après la séquence d'homologation.
depart	Séquence de démarrage du robot : – On attends que la tirette soit tirée – On sort de l'aire de départ et on attrappe la première balle
ramassage	Le robot ramasse des balles sur le terrain en utilisant la caméra
dechargement	Le robot choisit un panier, se dirige vers celui-ci et y décharge son barillet
pillage	Le robot choisit un panier, se dirige vers celui-ci et le vide de son contenu
defense	(N/A) Aurait pu contenir un code permettant de défendre les points acquis en empêchant le robot adverse d'accéder aux paniers.
gene	(N/A) Idem.
victoire	(N/A) Séquence à exécuter en cas de victoire pour dégouter l'adversaire.
defaite	(N/A) C'est pas bon pour le moral de l'équipe d'implémenter ce genre d'état, de toute manière il n'aurait jamais servi :) (eh oui la seule fois qu'on a perdu le robot était totalement bloqué).
ramassage_aveugle	Séquence à exécuter pour ramasser des balles dans le cas où on n'aurait pas de caméra.
ramassage_autour	Séquence permettant de ramasser les balles autour du terrain et de vider le barillet (en passant devant chacun des panier).

7.3.2 La transition entre les états

Une fois que chacun des super-états est défini, il faut définir comment passer de l'un à l'autre et dans quelles conditions on devrait choisir un état plutôt qu'un autre.

Pour chacun des états implémentés, on présentera vers quel état il peut aller et sous quelle condition.

arrete

8 Le séquenceur

Le séquenceur à été utilisé pour effectuer des tests d'intégration du robot. Il est composé de fonctions permettant de réaliser une suite de séquences définies et paramétrées dans un gros tableau.

Références

[1] F. de Lamotte. Les outils utilisés pour développer et débugger le code de la carte principale. http ://n00n.free.fr, 2002.

[2] F. de Lamotte. Mise en place d'une communication entre un 68332 et un pic pour dialoguer sur un bus i2c. http ://n00n.free.fr, 2002.