Les outils utilisés pour développer et tester le code de la carte principale

Florent de LAMOTTE

12 septembre 2002

Cette documentation présente les différents outils utilisés lors des phases de développement et de tests du code de la carte 68332.

Ces outils ayant évolué au cours du développement, ce document retrace aussi l'histoire de ces outils.

1 Motivations

Pour développer le code de la carte principale du robot Snooky, nous avons eu besoin d'outils de développement logiciels, dont la mise en place est présentée dans le document intitulé "développer sur 68332 avec les outils GNU" [1] dont nous allons présenter l'utillisation. Ceux-cis seuls n'étaient pas suffisant. Il nous a fallu développer des outils permettant de commander directement la carte principale et de diagnostiquer l'état du robot.

2 Les outils de développement

Cette section présente les différents outils permettant d'arriver à un programme fonctionnel et de le debugger.

2.1 La compilation du programme et l'exécution par la carte principale

2.1.1 Compilation assemblage et edition de liens : gcc, gas et ld

L'écriture du code en C doit suivre les conseils données dans le support de cours sur le C embarqué [2].

Pour compiler un programme C pour la carte principale, on utilise un compilateur GCC spécialement compilé pour le 68332[1]

On utilise aussi un *Makefile* qui nous permet de définir et d'automatiser le processus de compilation.

Dans ce Makefile, nous avons défini une règle de production, permettant d'obtenir un fichier objet au format coff : ".o" à partir d'un fichier source ".c" à l'aide de gcc.

```
%.o : %.c $(HEADERS)
        $(CC) -c $< $(CFLAGS)
```

Au début du Makefile, il faut définir le compilateur utilisé à l'aide de la variable *CC* et les options de compilation à l'aide de la variable *CFLAGS*. On a par la même occasion défini les commandes utilisées pour invoquer tous les outils.

[1] Le processus permettant d'obtenir des outils de developpement pour 68332 a été présenté dans le document "développer sur 68332 avec les outils GNU" [1].

```
CC = m68k-coff-gcc -m68332
LD = m68k-coff-ld
AS = m68k-coff-as -m68332
AR = m68k-coff-ar
OBJCOPY = m68k-coff-objcopy
RANLIB = m68k-coff-ranlib
RM = rm -f
TAS = tas

CFLAGS += -g -I./libESEO -Wall
C2SFLAGS += -I./libESEO -Wall
LDFLAGS = -L./libESEO
```

Nous avons aussi défini une règle de production permettant d'obtenir un fichier objet à partir d'un fichier source assembleur au format gas[2].

```
%.o : %.s
        $(AS) -o $@ $<
```

Il nous faut ensuite obtenir, à partir des différents fichiers objets dont nous disposons. Ceci se fait à l'aide de l'éditeur de liens *ld*.

Nous allons créer deux exécutables différents :

- Le premier sera un fichier *coff* et sera utilisé par gdb pour être mis en RAM puis exécuté. Pour produire ce fichier, l'éditeur de liens utilisera le fichier *ldscript*.
- Le second sera un fichier au format *S-record*, le format d'envoi de code binaire propre à Motorolla[3]. Ce fichier sera utilisé pour envoyer le code dans les mémoires flash et donc pour être utilisé lorsque le robot sera en autonome. L'éditeur de liens utilisera cette fois ci le fichier *ldscript_flash*.

```
# construit pour etre envoye a gcc
snooky_brain.coff : ldscript crt0.o snooky_brain.o $(OBJECTS)
        $(LD) -T ldscript $(OBJECTS) snooky_brain.o -lgcc  -lm -lc /usr/local/lib/gcc-lib/m68k-co

# code a envoyer en rom
rom.coff : ldscript_flash crt0.o snooky_brain.o $(OBJECTS)
        $(LD) -T ldscript_flash $(OBJECTS) snooky_brain.o -lgcc  -lm -lc /usr/local/lib/gcc-lib/m

#  S-Record pour envoyer le code a partir d'un flasheur
rom.srec : rom.coff
        $(OBJCOPY) -O srec -S rom.coff rom.srec

# Image binaire du code
rom.bin : rom.coff
        $(OBJCOPY) -O binary -S rom.coff rom.bin
```

[2] le format d'assemblage gas est legèrement différent du format d'assemblage 68000. Il permet d'avoir un format très peu différent suivant la plateforme. Le format gas est surtout le format des fichiers assembleur produits par gcc

[3] Intel lui utilise le format hex

Les fichiers utilisés par l'éditeur de liens lui permettent de placer les différentes sections du programme (.text, .data et .bss) aux bons endroits dans le fichier objet résultant. Il définit aussi l'emplacement de ces sections une fois que le programme sera lancé.

Nous donnons ici l'exemple de *ldscript_flash*, décrivant à ld comment constituer un code objet permettant ensuite de charger le code après l'avoir mis en flash.

```
STARTUP(crt0.o)
OUTPUT_ARCH(m68k)
OUTPUT(rom.srec) /* nom par defaut */

SEARCH_DIR(/usr/local/m68k-coff/lib/mcpu32);
SEARCH_DIR(/usr/local/lib/gcc-lib/m68k-coff/3.0.2/mcpu32);

/*
 * Point d'entree pour gdb
 */
ENTRY (gdb_startup)

/*
 * Un symbole pour definir l'emplacement de la pile
 * Ce symbole est utilisee par la routine sbrk,
 * chargee d'agrandir le tas
 */
__s_stack = 0x7F800;

/*
 * Emplacement de la table d'exception en RAM
 */
__s_ex_table = 0x40000;
__e_ex_table = 0x40400;

SECTIONS
{
        /*
         * Section .text (code)
         * Elle est placee dans la Flash et y reste
         */
        .text   0x00000000 : AT (0)
        {
                __s_text = . ;  /* definition de symboles */
                *     (.text)
                CONSTRUCTORS
                __e_text = . ;  /*  utilises dans le code d'initialisation*/
        } > rom

        /*
         * Section .data
         * Placee juste apres .text dans la FLASH
```

```
         * pour etre copiee apres la table d'exception dans la RAM
         * lors de l'initialisation @0x20400
         */
        .data   __e_ex_table : AT (SIZEOF(.text))
        {
                __s_data = . ; /* Ces symboles permettent de savoir */
                *(.data)
                __e_data = . ; /* ou copier .data dans la RAM */
        } > ram

        /*
         * Section .bss (variables globales non initialisee)
         * Elle est placee dans la ram, juste apres .data
         */
        .bss    __e_ex_table + SIZEOF(.data) :
        {
                __s_bss = . ; /* Ces symboles permettent */
                *(.bss)
                *(COMMON)
                __e_bss = . ; /* d'effacer le bss */
        } > ram
}

/*
 * Definit la configuration memoire
 * Permet a ld de faire quelques petites verifications
 */
MEMORY {
  rom : ORIGIN = 0x00000000, LENGTH = 256K
  ram : ORIGIN = 0x00040000, LENGTH = 256k
}
```

On remarque que pour chaque section, on bien défini l'adresse à laquelle elle sera placée pendant l'exécution du programme et l'adresse qu'il occupera en flash (spécifiée à l'aide de *AT*).

Pour la section *.bss* on n'a pas défini d'emplacement en ROM puisque *.bss* est la section où sont placées par gcc toutes les variables globales et statiques non initialisées (prenant par défaut la valeur 0). Cette section n'a pas de place dans le code objet puisqu'elle prendrait de la place inutile. Elle doit par contre être initialisée au démarrage du programme par le code de démarrage (*crt0.s*) qui a donc besoin des symboles *__s_bss* et *__e_bss*.

2.1.2 Chargement du programme en RAM à l'aide du debugger gdb

Le débugger, gdb à été utilisé non seulement en tant que debugger, mais aussi pour charger le programme directement en RAM.

Charger le programme directement en RAM à travers le BDM[4] prends beaucoup moins de temps que de le flasher. Cela permet en plus une fois le code chargé de le debugger.

[4]Background Debugger Mode, le BDM permet de charger directement du code en RAM et de debugger le 68332 diremectement à partir d'un PC

La documentation de Pavel Pisa sur le BDM [3] est très complète et donne un maximum d'informations au sujet de l'utilisation du BDM sous linux.

Pour fonctionner correctement, les outils utilisant le BDM[5] utilisent le fichier *cpu32init*.

```
# on eteinds le watchdog
W 0xFFFA21 0x0 1

# on ajuste la vitesse du processeur
W 0xFFFA04 0x7F00 2

# initialisation des chip-selects
W 0xFFFA44 0x00FF0000 4
W 0xFFFA48 0x00057870 4
W 0xFFFA4C 0x04055830 4
W 0xFFFA50 0x04053830 4
```

Ce fichier permet d'initialiser certains registres du 68332. Il sert par exemple à initialiser les Chip-Selects. En effet, si les Chip-Selects ne sont pas initialisés, il n'est pas possible d'accéder à la mémoire.

La syntaxe est très simple et est décrite dans [3]. Ici, par exemple, la commande *W* demande une écriture à une adresse mémoire (premier paramètre) de la valeur passée comme deuxième paramètre. Le troisième paramètre quant-à-lui spécifie la taille du transfert.

L'initialisation du *BDM* est très simple et systématique. Elle a donc été placée dans un script pour gdb, appellé *icd.gdb*[6].

```
target bdm /dev/icd_bdm0
bdm_setdelay 0
bdm_reset
bdm_autoreset off
```

La ligne de commande permettant de lancer gdb pour debugger la cible *snooky_brain.coff* sera donc la suivante :

```
m68k-bdm-coff-gdb -x icd.gdb snooky_brain.coff
```

Le chargement puis le lancement du programme se fait ensuite à l'aide de la commande *run*. Gdb va alors vous demander si vous voulez charger le programme en mémoire, il vous suffit à ce moment de répondre par l'affirmative pour que le programme soit effectivement chargé en mémoire puis exécuté.

2.1.3 Flashage à l'aide de bdm-load

Le programme *bdm-load* permet de contrôler le bdm. parmi ses fonctions, on en trouve une qui lui permet de flasher directement des flash.

Les flash utilisées dans le cadre de la carte principale du robot Snooky étaient deux AMD AM29F010 configurées pour utiliser un bus 16bits. Nous avions de la chance puisque bdm_load supporte effectivement cette configuration.

Bdm-load, tout comme gdb, utilise le fichier cpu32init pour initialiser le 68332 avant de procéder à l'envoi du code.

[5] gdb et bdm-load

[6] On demande à gdb d'exécuter un script à l'aide du commutateur -x

Pour faciliter l'envoi du code vers la cible, nous avons écrit une règle dans le Makefile lançant directement la commande de chargement. Cette règle est la règle *load*.

```
load : rom.srec
        bdm-load -f auto@csboot -c -r -e -s -l rom.srec
```

Expliquons les différents commutateurs utilisés :

– le commutateur -f permet de spécifier sur quelle flash on veut envoyer le programme en donnant son type. Ici, on demande à bdm-load d'envoyer le code sur *csboot* et de détermeiner le type de flash que l'on utilise. Le fait de demander à bdm-load de déterminer automatiquement le type de flash permet de s'assurer que la flash fonctionne bien.
– le commutateur -c oblige bdm-load à tester la flash
– le commutateur -e demande à bdm-load d'effacer la flash avant le transfert
– le commutateur -s passe bdm-load en mode script, il se termine donc après avoir exécuté toutes les commandes
– le commutateur -l demande à bdm-load de charger le fichier spécifié en argument dans la flash

Une documentation des actions effectuées par bdm-load peut être obtenue en utilisant le commutateur -h. De plus le document de Pavel Pisa [3] décrit assez bien comment utiliser bdm-load.

2.2 Le debuggage du programme

Le debuggage du code de la carte s'effectue à l'aide de gdb ou de son interface graphique ddd. Ddd est principalement utilisé pour les opérations plus lourdes.

Pour lancer ddd en utilisant la version de gdb compilée pour utiliser le bdm on utilise la ligne de commande suivante :

```
ddd --debugger m68k-bdm-coff-gdb -x icd.gdb snooky-brain.coff
```

3 Les outils de commande et de diagnostic pour la carte

3.1 L'ère de Minicom

Le premier organe que nous avons essayé de faire fonctionner sur la carte principale a été le port série.

En effet, l'interface série est un périphérique interne au 68332, ne demandant que très peu de connaissances pour être mis en oeuvre. De plus, cette interface série, nous l'avions déjà utilisée lors des miniprojets microprocesseurs en I1. Il était donc normal d'essayer d'écrire du code capable de nous "parler" au travers de cette liaison.

Le développement du code pour la carte principale étant réalisé sous Linux, nous avons choisi d'utiliser le programme minicom, équivalent à l'hyperterminal de Windows.

Les premiers programmes se limitaient à envoyer des données à minicom au travers de la liaison série, d'abord de simple caractère, des chaînes de caractères ! ! ! Ces chaînes correspondaient à des résultats de traitements que l'on faisait éxécuter sur la carte.

Très vite on à ajouté à cette possibilité de visualisation de l'état de la carte la possibilité d'envoyer des ordres à partir du terminal. Ces ordres correspondaient en fait à l'envoi d'un caractère à l'aide du clavier. Le code de la carte principale s'occupait en fait à l'aide d'un gros switch, d'interpréter les ordres et de les exécuter.

3.2 La carte sur laquelle on branche la tirette

C'est bien beau tout ca mais quand on debranche le PC, après on voit plus rien. Il était nécessaire de disposer d'une carte permettant de visualiser l'état du robot et de paramétrer son fonctionnment. Et oui, c'est la naissance de la carte de commande de la carte de commande, la fameuse carte sur laquelle on branche la tirette :).

Sur cette carte, nous avons disposé quatre leds, un bornier dip, et une prise pour brancher la tirette.

Quand il fallut tester les déplacements du robot, nous avons mis en place un séquenceur. Nous disposions d'un gros tableau "en dur", dans le code principal, contenant des séquences de commandes avec leurs paramètres. Les différentes séquences pouvaient être séléctionnées à l'aide du bornier et le lancement de la séquence s'effectuait soit à l'aide de la tirette (qui n'était même pas une vraie tirette à l'époque), soit à l'aide d'une commande sur le terminal.

Les commandes du séquenceur étaient constituées non seulement de commandes de déplacement du robot mais aussi de commandes de visualisation à l'aide des quatre leds. Ainsi il nous était possible de savoir où en était le robot dans sa séquence. Une fois que la méca à été prête, nous avons ajouté des commandes de gestion de la carte mécanique. Le robot fonctionnait en autonome.

Le code de ce séquenceur n'est plus compilé avec le reste du code. Les fichiers source correspondant sont par contre toujours là, dans les fichier *sequenceur.h* et *sequenceur.c*.

3.3 Chef la camera n'est pas encore prête

Lorsqu'il fallut songer à implémenter le code gérant la caméra, l'interface avec la carte principale n'était pas encore prête.

La caméra utilisant une liaison série pour dialoguer avec la carte principale, nous avons décidé d'écrire un programme simulant cette caméra sur PC.

Le problème posé par cette liaison est qu'elle utilise la liaison série de la carte principale, déjà utilisée pour le débuggage.

Le programme *simu_cam*, qui se limitait à envoyer une information concernant la position des balles à la carte principale s'est donc vu adjoindre une console, remplaçant le programme *minicom*, jusqu'alors utilisé pour remplir cette fonction.

Nous y avons aussi ajouté des boutons permettant d'invoquer rapidement des commandes de la carte principale.

La figure 1 présente la fenêtre principale du programme *simu_cam*. Celle-ci est composée de deux console, l'une affichant des messages interprétés par le programme, l'autre, ceux que celui-ci n'a pas pu interpréter. La fenêtre principale possède aussi une barre de boutons située sur la gauche de l'écran.

Pour faciliter le debuggage, nous avons ajouté un terrain virtuel dans une fenêtre séparée.

Cette fenêtre, visible sur la figure 2 permet de placer des balles pour tester l'envoi des coordonnées de celles-ci dans la carte principale. Le placement se fait soit manuellement, soit de manière aléatoire. Au cours des développement, nous avons ajouté la possibilité de visualiser la position de notre robot et du robot adverse sur le terrain virtuel.

Dernière amélioration apportée au processus de test. Pour faciliter ceux-ci, nous avons ajouté au programme de la carte principale la possibilité d'envoyer des informations de debuggage vers le PC. Ces informations peuvent être envoyées pendant l'interruption timer. Grâce au bornier, on peut choisir d'empêcher l'envoi de ces informations pendant l'interruption, ce qui évite de polluer la communication avec la caméra (quand on mets la vraie caméra).

La figure 3 est une capture d'écran du panneau affichant les informations en provenance de la carte principale.

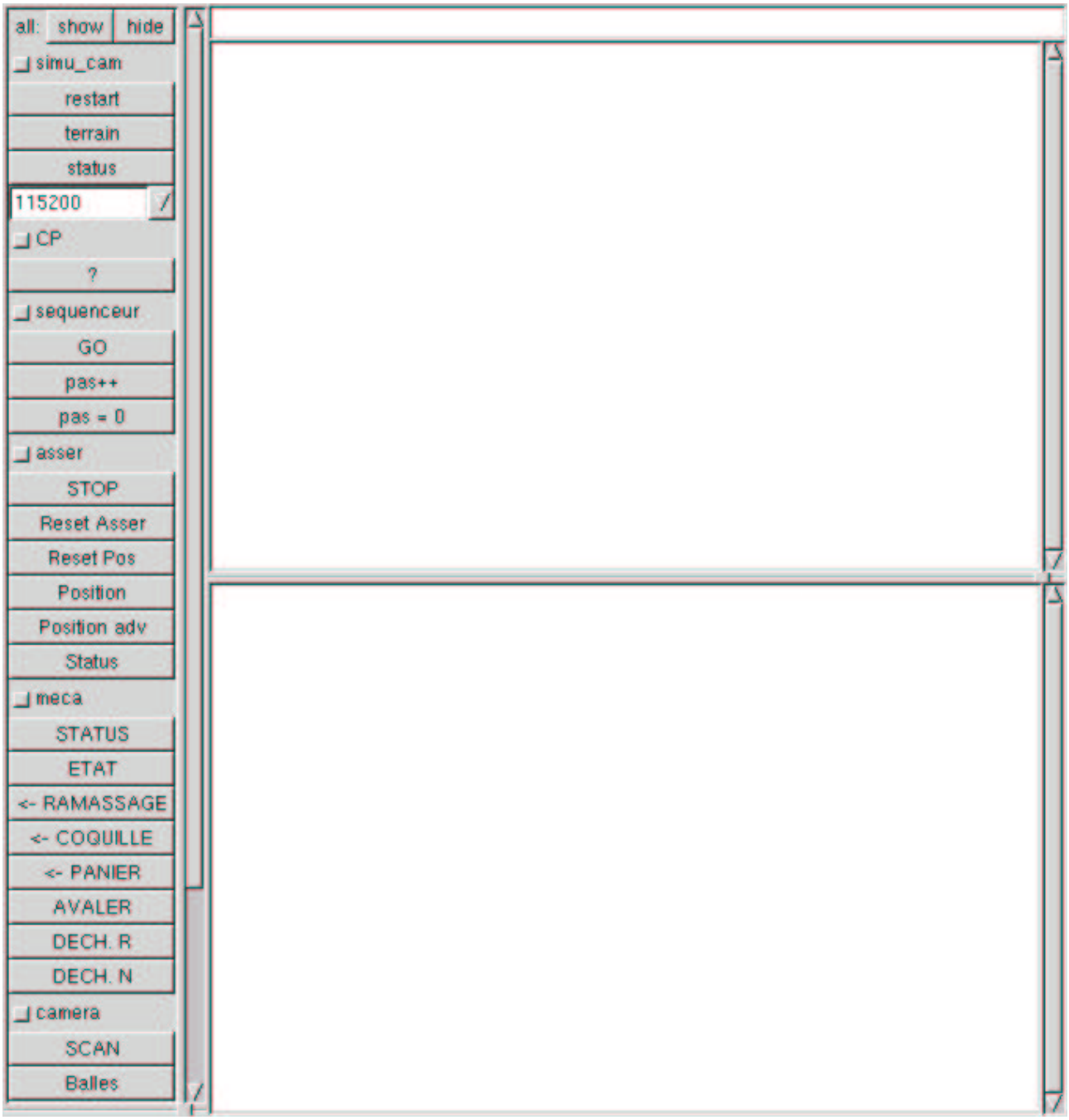

FIG. 1 – Fenêtre principale de simu_cam

Références

[1] F. de Lamotte. Développer en c sur 68332 avec les outils gnu, 2002.

[2] R. Perdriaux. Introduction au langage c embarqué. ESEO, 2001.

[3] P. Pisa. Bdm interface for motorola 683xx mcu usage with gdb debugger, 2000.

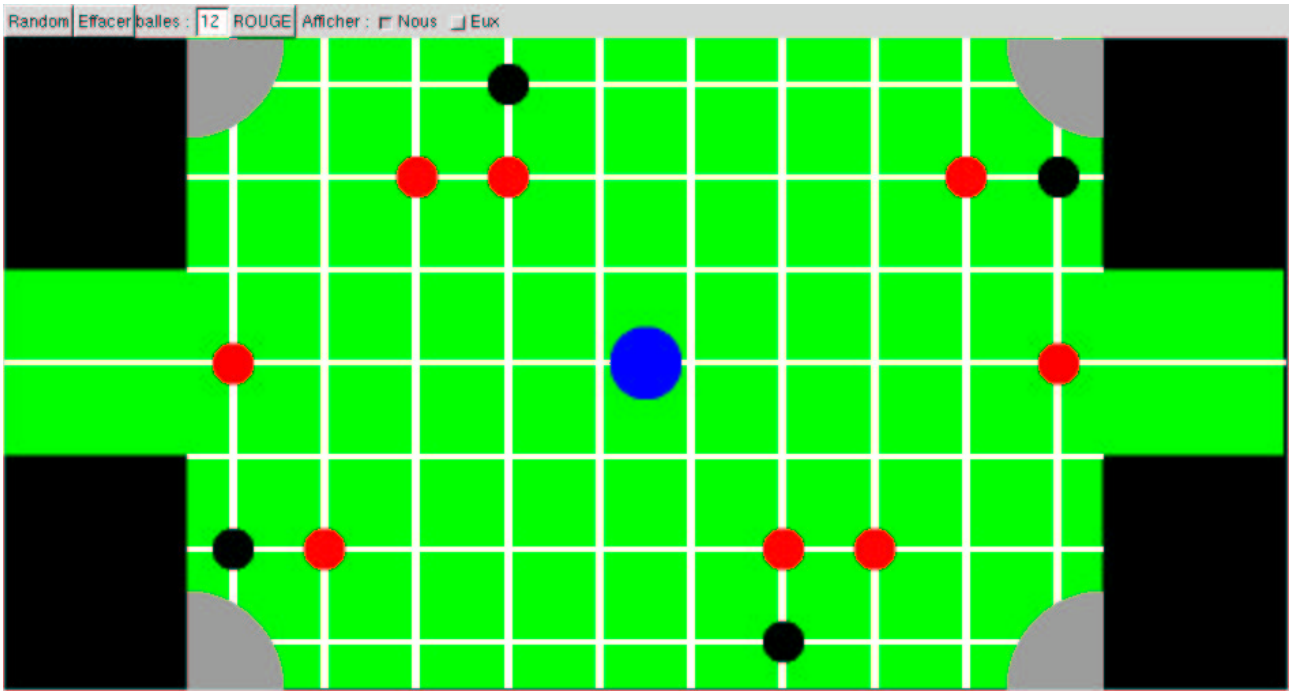

FIG. 2 – Représentation du terrain

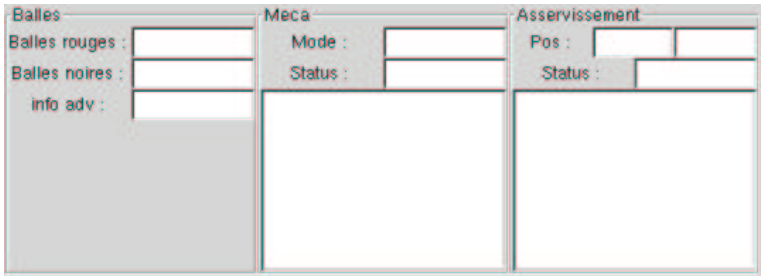

FIG. 3 – Panel affichant les informations sur le robot