# Introduction à la programmation des ERS-7 avec le SDK R-Code

Version anglaise originale par Ricardo A. Téllez, v1.2 (4 septembre 2004)
Version française par TME (4 mai 2005)

**Version du 20/06/2005**
Remerciements à
**laulau_q, robintoc, Sp@rK, virginiela**
du forum http://www.premiumwanadoo.com/aibo-fr.com/forum/index.php

**Index**

**1 – Introduction**

**2 - Le SDK pour R-Code**
- 2.1 – Description
- 2.2 – Installation
  - 2.2.1 – Préparation du Memory Stick
  - 2.2.2 – Exécution d'un programme écrit en R-Code
  - 2.2.3 – Paramétrage de la console en Wi-Fi
- 2.3 – Considérations d'ordre général
- 2.4 – Programmer en R-Code
- 2.5 – Les commandes R-Code
- 2.6 – Les opérateurs logiques simples
- 2.7 – Les variables système
- 2.8 – Les actions connues par AIBO
- 2.9 – Les expressions reconnues par AIBO
- 2.10 – Chasser les bugs

**3 – Exemples**
- 3.1 – Le programme d'exemple « Greeting.R »
- 3.2 – Le programme d'exemple « Maze.R »

**4 – Création et utilisation de fichiers de ressources**
- 4.1 – Créer des fichiers de ressources
  - 4.1.1 – Créer des fichiers de mouvements
  - 4.1.2 – Créer des fichiers audio
  - 4.1.3 – Créer des animations IllumFace
- 4.2 – Conversion en fichier ODA
- 4.3 – Créer le fichier MWC
- 4.4 – Copier les fichiers sur le Memory Stick

# 1 – Introduction

Le SDK R-Code est un ensemble d'outils mis gratuitement à disposition par SONY dans le but de permettre la programmation d'AIBO en utilisant le langage de script R-Code. Le terme script désigne un fichier texte tout simple contenant des commandes toutes simples qui permettent au programmeur d'envoyer des instructions vers AIBO à un niveau plus élevé du système qu'en utilisant un environnement de développement en C++ (OPEN-R par exemple). Les langages de script ont l'avantage d'être faciles à apprendre et de ne pas nécessiter de compilation, mais en contrepartie ils offrent un contrôle moins élevé au programmeur sur la façon dont les choses se déroulent. Vous pouvez voir ici un exemple écrit en R-Code :

```
:1000
PLAY:ACTION:STAND
IF:Face:=:1:THEN
WAIT
PLAY:ACTION:CLIFF_DETECT_OFF
WAIT
```

R-Code permet de programmer des choses compliquées (comme marcher ou dancer) à partir de quelques commandes dans un fichier texte. Considérez les instructions R-Code comme étant des macros et les programmes OPEN-R comme étant le programme de ces macros. Vous ne pouvez pas obtenir plus de précision en utilisant des macros que ce que permet le programme OPEN-R qui les compose. Utilisez R-Code quand vous n'avez pas besoin d'avoir un contrôle précis sur le chien, juste pour faire quelques actions ou lancer quelques mouvements et réponses prédéfinis. Mais ne vous méprenez pas : R-Code est un outil puissant qui permet l'implémentation de fonctions complexes.

**Pour pouvoir utiliser R-Code avec AIBO, vous allez essentiellement avoir besoin de deux choses :** Premièrement, un **Memory Stick (référence ERA-MS008 ou ERA-MS016)** correctement configuré, deuxièmement, votre **programme écrit en R-Code.** Pour préparer le Memory Stick, il va vous en falloir un vierge sur lequel vous allez copier quelques fichiers particuliers fournis par SONY (les fichiers Redist7[1]). Ces fichiers sont appelés *Système de base* et font partie du SDK pour R-Code. Une fois que vous avez votre Memory Stick avec le système de base installé dessus, vous pouvez y transférer votre programme en R-Code dessus.

Image 1 : Un Memory Stick ERA-MS008 destiné spécialement à la programmation d'AIBO.

Bien que le SDK pour R-Code contienne quelques fichiers destinés spécifiquement aux systèmes tournant sous Microsoft Windows (Rtool et NsmAuth), le développement de programmes en R-Code peut se faire sur tout système d'exploitation disposant d'un éditeur de texte. Les programmes écrits en R-Code sont de simples fichiers texte qui ne nécessitent aucune compilation, de sorte que tout OS disposant d'un éditeur de texte fait l'affaire.

Ce que vous pouvez faire avec R-Code :

_ Faire prendre à AIBO ses trois postures de base (assis, debout et couché),
_ Faire marcher, tourner, taper, toucher, bouger sa tête et suivre sa balle rose à AIBO,
_ Faire trouver sa balle rose, reconnaître un visage et son AIBOne à AIBO,
_ Faire reconnaître 53 commandes vocales à AIBO,
_ Exécuter 600 éléments intégrés (mouvements, sons, animations IllumFace) fournis avec les AIBOware commerciaux (AIBO MIND),
_ Exécuter des mouvements créés avec le logiciel AIBO Motion Editor (MEdit),
_ Jouer des fichiers sons MIDI ou WAV créés par les utilisateurs,
_ Exécuter des animations IllumFace créées par des utilisateurs,
_ Collecter des informations relatives à la détection d'obstacles (calcul de distance) et vérifier l'état des capteurs de la tête, du dos et de la bouche,
_ Utiliser des commandes de branchement IF et des boucles FOR,
_ Exécuter des sous-routines,
_ Utiliser des variables entières codées sur 32 bits,
_ Effectuer des calculs,
_ Faire des PUSH et des POP sur la Pile,
_ Utiliser le système Wi-Fi d'AIBO pour lui envoyer des commandes et faire du debugage.

[1]Redist7 est le paquetage destiné aux AIBO modèle ERS-7. Les autres modèles recquiérent un autre système R-Code de base, également fourni par SONY. Nous nous focalisons ici sur la version de R-Code destinée aux ERS-7.

# 2 – Le SDK pour R-Code

## 2.1 – Description

Le SDK pour R-Code est l'ensemble des fichiers fournis par SONY qui permettent aux utilisateurs de programmer en R-Code. Il inclut plusieurs éléments :

_ Redist7 : C'est l'ensemble des fichiers qui constituent le système de base du Memory Stick. Ce système de base va être nécessaire pour exécuter des programmes R-Code. Redist7 contient les versions Japonaise et Anglaise, vous allez donc devoir choisir celle qui vous convient.

_ Rtool : Il s'agit d'une application Windows qui convertit les fichiers MTN (fichiers décrivant les mouvements d'AIBO, créés à l'aide de MEdit ou un autre éditeur de mouvements) dans un format reconnu par R-Code (ce format est nommé ODA). Cet utilitaire crée également un fichier spécial nommé ERS-7.MWC qui contient une liste de mouvements, animations IllumFace et sons. Nous parlerons de la génération de tels fichiers plus tard. Un script R-Code peut appeler n'importe quel élément contenu dans un fichier ODA en utilisant la commande suivante : *PLAY:MWCID:<numero>*.

_ NsmAuth : C'est une bibliothèque de fonctions utilisée par les programmes Windows qui veulent accéder à la ligne de commandes Wi-Fi[2] d'AIBO avec un système d'authentification.

## 2.2 – Installation

Allez sur le site de SONY dédié à OPEN-R (http://openr.aibo.com) et téléchargez les fichiers relatifs à R-Code. Cela inclut Redist7_ver2.zip (du 21/02/2005), RTool_ver1.zip (du 17/05/2004) et NsmAuth_ver1.zip (du 17/05/2004 également). Sachez tout-de-même que vous n'aurez réellement besoin que du premier fichier, les deux autres n'étant que des compléments pour des environnements de travail plus sophistiqués.

### 2.2.1 – Préparation du Memory Stick

Il faut tout d'abord préparer le Memory Stick avec la bonne structure de fichier. Ainsi, le Memory Stick sera à même de stocker vos programmes en R-Code. Décompressez Redist7_ver2.zip. Cela génére plusieurs fichiers et dossiers. Maintenant, copiez le répertoire Redist7/Eng/OPEN-R sur un Memory Stick vide : Le voilà prêt à recevoir vos créations.

### 2.2.2 – Exécution d'un programme écrit en R-Code

Créez votre programme avec n'importe quel éditeur de texte. Sauvegardez-le sous le nom R-CODE.R (ce nom est obligatoire). Ensuite, installez ce fichier dans le dossier /OPEN-R/APP/PC/AMS/ du Memory Stick. Insérez le Stick dans AIBO et allumez-le : Il va automatiquement exécuter votre programme.

### 2.2.3 – Paramétrage de la console en Wi-Fi

La console Wi-Fi du SDK pour R-Code permet à l'utilisateur d'envoyer des commandes directement à AIBO à partir d'une session TELNET. Cela a de multiples avantages comme le debugage en ligne ou l'essai de routines et de commandes. Suivez cette procédure pour paramétrer la console :

1. Activez le réseau Wi-Fi. Cette manoeuvre est expliquée sur le site web dédié à OPEN-R (http://openr.aibo.com) ou dans le *Aibo quick-start guide* écrit par Ricardo A. Téllez (disponible sur http://www.ouroboros.org/~rt71592 [EN]). En fait, vous allez devoir changer les paramétres de réseau du fichier /OPEN-R/SYSTEM/CONF/WLANDFLT.TXT dans le Stick. Veuillez vous référer aux documentations adéquates en cas de souci.

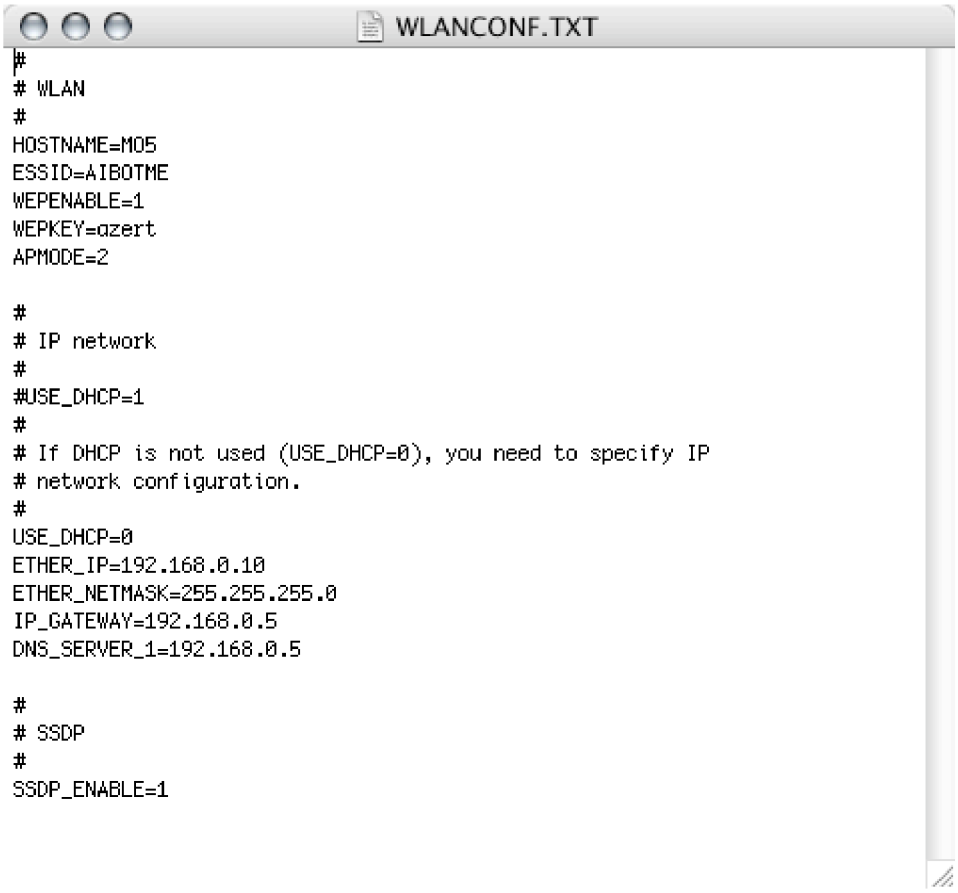


Image 2 : Mon fichier OPEN-R/SYSTEM/CONF/WLANDFLT.TXT (ça peut aider).

2. Désactivez l'authentification pour R-Code. En faisant cela, ni login, ni mot de passe ne seront demandés lors de la connexion avec AIBO. Pour réaliser cette opération, effacez le fichier /OPEN-R/APP/DATA/P/OWNER.TXT et créez /OPEN-R/APP/PC/AMS/NOAUTH.CFG (n'inscrivez rien dans ce fichier, il doit rester vide et avoir une taille de 0 octets).

Image 3 : Mon fichier /OPEN-R/APP/DATA/P/OWNER.TXT

3. Insérez le Memory Stick dans AIBO et allumez-le.

4. Connectez-vous à AIBO en utilisant le protocole Telnet. Les connexions sont acceptées sur le port 21002, la ligne de commandes à saisir est donc de la forme *telnet IP_de_votre_AIBO 21002* (chez moi, ça donne « telnet 192.168.0.10 21002 »). Si la connexion fonctionne, vous verez apparaître le message d'accueil de R-Code suivi de quelques informations et du prompt sur votre écran (voir image 4).

5. Vous avez la possibilité d'envoyer des commandes directement à AIBO en les saisissant dans la console.

6. Vous pouvez envoyer des programmes complets en saisissant la commande *EDIT*, puis en tapant vos instructions dans la console. Une fois que vous avez fini, saisissez *END* pour indiquer que votre programme est terminé et prêt à être exécuté. Pour exécuter le programme, tapez *RUN*. Tout programme envoyé de cette manière remplace celui qui est déjà présent dans la mémoire d'AIBO (et non pas dans le Memory Stick).

7. Pour vous déconnecter du robot, tapez *@DISS*.

[2]La ligne de commandes Wi-Fi sera détaillée dans la section 2.5

## 2.3 – Considérations d'ordre général

Avant de commencer à programmer avec R-Code, quelques mises au point s'imposent :

1. Les scripts R-Code sont composés de lignes de texte indiquant des commandes. Les commandes sont des mots séparés par « : ». Seuls les caractères ASCII et les underscores (« _ ») peuvent être utilisés dans les mots et les lignes ne peuvent pas faire plus de 127 octets de long (soit 127 caractères, puisque 1 caractère=1 octet) en comptant les commentaires et le caractère de saut de ligne (chaque ligne de texte se termine par un caractère invisible indiquant à l'ordinateur qu'il faut aller à la ligne). Les caractères d'indentation (tabulations et espaces en début de ligne) ne sont pas pris en compte.

2. Les lignes commençant par « : » sont considérées comme des labels. Ils sont utilisés pour sauter d'une partie du programme à une autre en utilisant la fonction *GOTO*. Les lignes qui débutent par un caractère non-alphanumérique ou différent de « : » sont traitées comme des commentaires. Un commentaire peut aussi être indiqué avec un double slash (« // »). Tout ce qui est après le double slash est vu comme un commentaire.

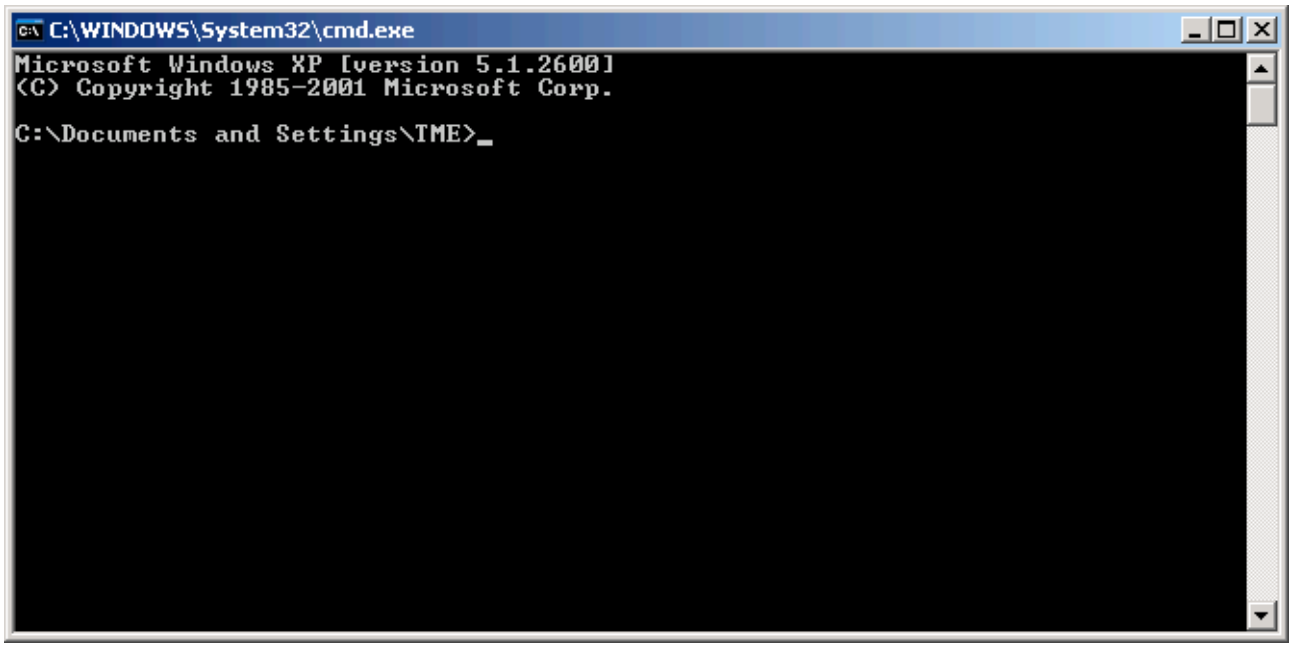


Image 4 – 1 : Sous Windows, on lance le programme CMD.EXE

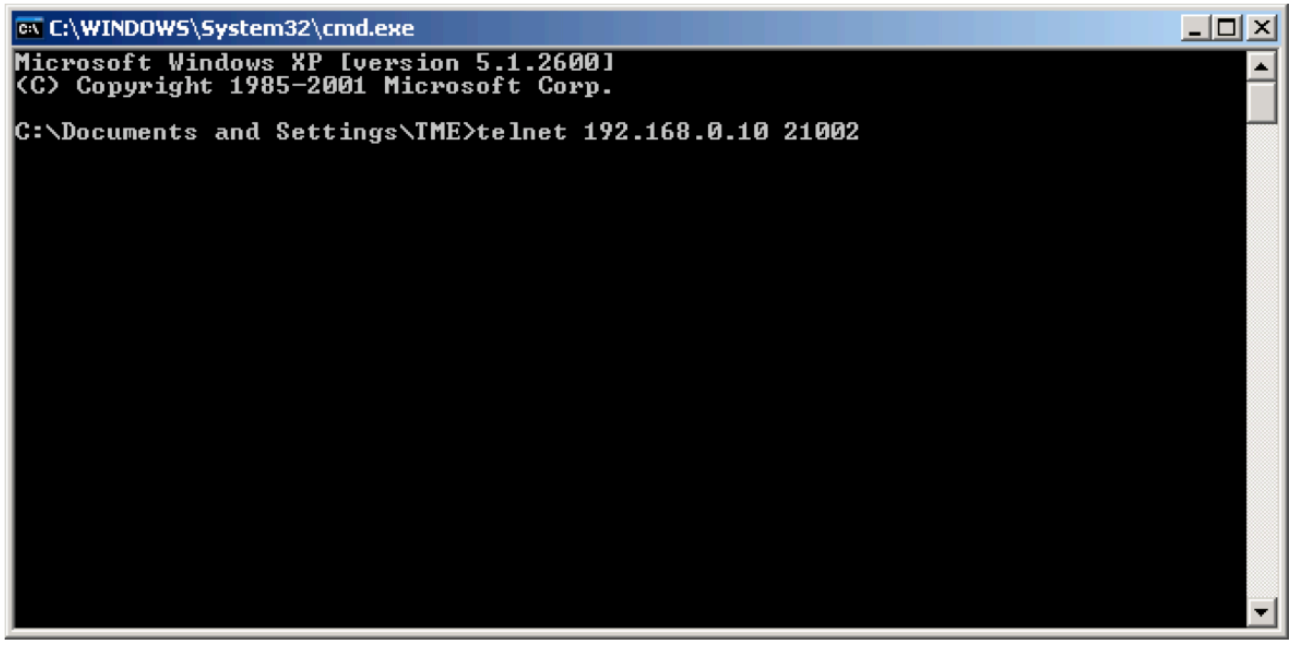


Image 4 – 2 : On saisit la ligne de commandes qui va bien.

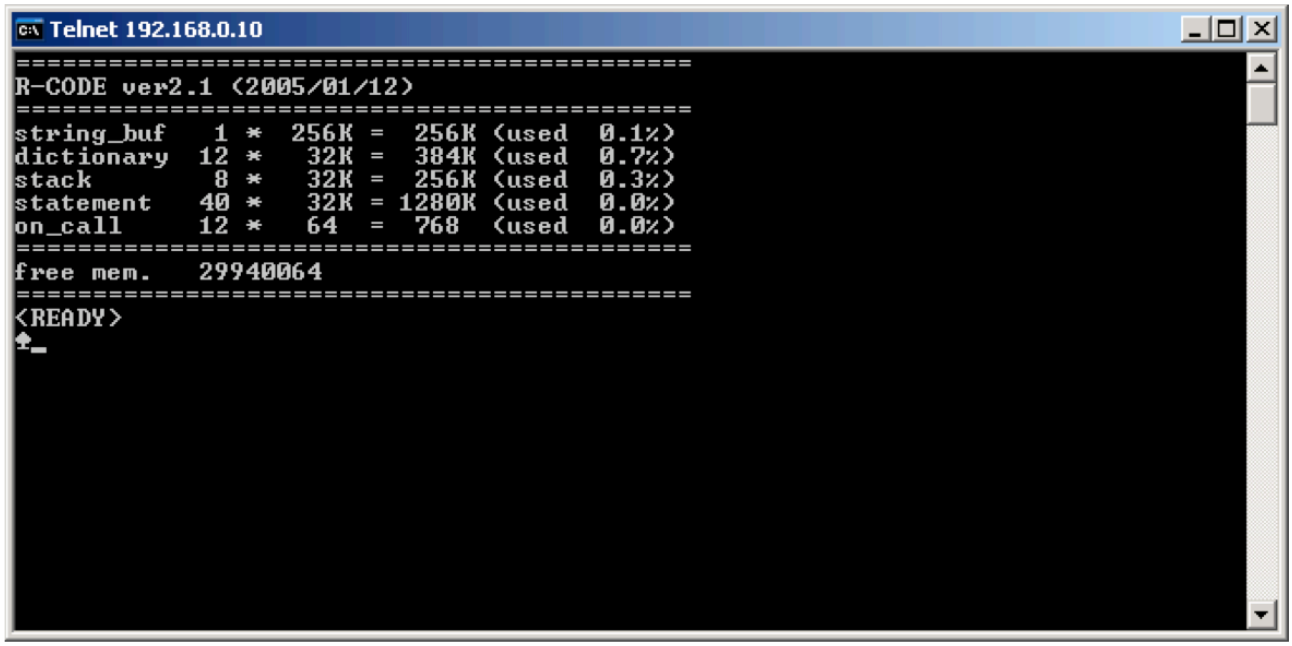


Image 4 – 3 : La session TELNET avec AIBO est ouverte. On peut saisir les instructions R-Code.

3. R-Code est **case sensitive**. Cela signifie que *Value* et *VALUE* sont des variables différentes. Sachez également que les mots en majuscule sont réservés pour R-Code, il est ainsi déconseillé de donner des noms en majuscule à vos propres variables. Par exemple, *Value* et *value* sont des noms corrects pour vos variables, mais *VALUE* n'est pas bon.

4. Les mots commençant par « + », « - » ou un chiffre sont traités comme des nombres. Les nombres débutant par 0x ou 0X sont considérés comme des valeurs

exprimées en hexadécimal, ceux débutant par 0o ou 0O sont considérés comme des valeurs exprimées en octal et ceux qui commencent par 0b ou 0B sont considérés comme des valeurs exprimées en binaire.

5. Seuls des entiers sur 32 bits (donc codés sur 4 octets, vu qu'un octet=8 bits) peuvent être utilisés.

## 2.4 - Programmer en R-Code

Tout programme R-Code est une liste de commandes pour le robot AIBO. Les programmes ressemblent fortement à du BASIC et comportent toutes les instructions classiques, comme les conditionnelles, les sous-routines, les boucles, l'utilisation de variables, etc, mais R-Code dispose également d'une Pile qui fonctionne comme celle qu'on utilise lorsqu'on fait de l'Assembleur. La liste des commandes disponibles est donnée à la section 2.5 et est très explicite.

Habituellement, le programme en R-Code est créé en utilisant un éditeur de texte ASCII (Ex. : Notepad (PC), TextEdit (Mac),...). Le programme doit être saisi dans l'éditeur de texte et sauvegardé sous le nom R-CODE.R sur le Memory Stick dans /OPEN-R/APP/PC/AMS/ une fois achevé. Ensuite, le Stick est inséré dans AIBO et le programme se lance dès que le robot est allumé. Il y a la possibilité de créer le programme directement dans AIBO en utilisant la console Wi-Fi. Cette méthode a l'avantage d'être plus pratique et rapide, notamment pour faire la chasse aux bugs (voir section 2.10).

R-Code met à disposition un ensemble de variables qui permettent au programmeur de connaître l'état du robot. La liste exhaustive de ces variables est donnée dans la section 2.7. Il y a des variables qui correspondent aux valeurs d'éléments internes d'AIBO, mais il y en a aussi qui indiquent les valeurs relevées par ses capteurs. Le programmeur peut consulter et modifier le contenu de ces variables à n'importe quel moment et faire agir AIBO en conséquence. Par exemple, une variable appelée *Face* indique quand AIBO a reconnu un visage (la variable est alors passée à 1). Vous pouvez utiliser cette valeur pour savoir si AIBO est en face de quelqu'un. D'autres variables existent qui montrent par exemple l'état du capteur de la patte ou la détection de la AIBall ou du AIBone. Le programmeur peut créer ses propres variables et les utiliser pour effectuer des calculs ou les passer en paramètres à des sous-routines (en utilisant la Pile). Mettez les paramètres sur la Pile en utilisant la commande *PUSH*, puis appelez la sous-routine (grâce à la commande *CALL*) et, une fois à l'intérieur, récupérez les paramètres dans l'ordre où ils ont été enregistrés à l'aide de la commande *ARG*.

R-Code permet de surcroît de faire exécuter des actions à AIBO. Une action est un mouvement (ou un groupe d'actions) prédéfini identifié par un nom. Ces mouvements ont été pré-enregistrés par les ingénieurs de SONY, vous n'avez donc pas à les conçevoir. Vous pouvez spécifier quelle action vous voulez que le robot exécute simplement en l'indiquant, et le robot l'effectue. Saisissez *PLAY:ACTION:<l'action>* pour faire une action. Par exemple, vous pouvez ordonner à AIBO de s'asseoir (*SIT*) en tapant *PLAY:ACTION:SIT*. Certaines actions nécessitent plusieurs paramètres que vous devez ajouter à la commande. Par exemple, si vous voulez qu'AIBO tourne de quelques degrés, vous devez indiquer le nombre de degrés en paramètre (*PLAY:ACTION:TURN:30*). Une liste exhaustive de toutes les actions disponibles est fournie à la section 2.8, mais vous pouvez créer vos propres actions avec les logiciels Medit et Rtool (voir section 4).

Il est également envisageable de faire reconnaître quelques commandes vocales à AIBO

en utilisant R-Code. AIBO incorpore un système de reconnaissance vocale dont le programmeur peut se servir. Il a aussi un dispositif d'identification de tonalité et de son capable de reconnaître ceux émis par d'autres AIBO (ce qui permet de les faire communiquer facilement). La variable *AU_Voice_ID* indique l'ID du mot reconnu. Une liste des mots identifiables par AIBO est fournie à la section 2.9.

Pour résumer, tout programme R-Code va consister en une série de commandes qui vont faire exécuter des actions à AIBO. La plupart des actions et des éléments reconnus sont intégrés à R-Code, il ne reste ainsi plus au programmeur qu'à les exploiter. Les programmes R-Code peuvent être aussi complexes que vous le désirez !

## 2.5 - Les commandes R-Code

Voici la liste de toutes les commandes R-Code classées par fonction accompagnées d'une brève description de leur fonctionnement :

CONTROLE

**Nom**

*EDIT* – Charge un programme (Transfère le programme du PC à AIBO).

**Syntaxe**

*EDIT*
*< programme >*
*END*

**Paramètres**

Aucun

**Description**

Charge un programme R-CODE à partir de EDIT jusqu'à END dans la mémoire.
Tout programme déjà présent est effacé.

**Exemple**

```
EDIT
FOR:i:1:10
        PLAY:ACTION:xxx
        WAIT
NEXT
EXIT
END
```

---

**Nom**

*END* – Indique la fin du programme à transférer en mémoire.

**Syntaxe**

*EDIT*
*< programme >*
*END*

**Paramètres**

Aucun

**Description**

Voir ce qui expliqué pour la commande *EDIT*.

**Exemple**

```
EDIT
FOR:i:1:10
    PLAY:ACTION:xxx
    WAIT
NEXT
EXIT
END
```

---

**Nom**

*RUN* - Exécute le programme chargé en mémoire.

**Syntaxe**

*RUN*

**Paramètres**

Aucun

**Description**

Lance l'exécution du programme qui se trouve en mémoire.

**Exemple**

*RUN*

---

**Nom**

*EXIT* – Stoppe le programme.

**Syntaxe**

*EXIT*

**Paramètres**

Aucun

**Description**

Arrête l'exécution du programme.

**Exemple**

```
EDIT
FOR:i:1:10
    PLAY:ACTION:xxx
    WAIT
NEXT
```

*EXIT*
*END*

---

**Nom**

*HALT* – Stoppe le programme / Eteint AIBO.

**Syntaxe**

*HALT*

**Paramètres**

Aucun

**Description**

Arrête le programme en cours d'exécution, puis met AIBO hors tension.
L'extinction prend quelques secondes.

**Exemple**

```
EDIT
FOR:i:1:10
    PLAY:ACTION:xxx
    WAIT
NEXT
HALT
END
```

---

**Nom**

*INIT* – Initialise l'environnement logiciel R-CODE

**Syntaxe**

*INIT:<init_level>*

**Paramètres**

<init_level>  Voir description

**Description**

*INIT:0* Initialise la totalité du système R-CODE.
Cette opération est effectuée automatiquement au démarrage et n'a d'utilité que pour le système. Vous ne devez pas l'utiliser.

*INIT:1* Accès au mode debug.
Lance le debugger de R-CODE (actuellement développé à part).
Vous ne devez pas l'utiliser.

*INIT:2* Initialise le fichier des mot-clefs et les variables.

*INIT:3* Cette phase d'initialisation est faite automatiquement lorsqu'une connexion Wi-Fi est établie.
Un second argument optionnel spécifie l'identifiant de l'AIBO concerné (AiboId).
Vous ne devez pas utiliser cette instruction dans vos programmes R-Code.

*INIT:4* Notification de déconnexion du réseau Wi-Fi (Rcomm->RMain)

*INIT:9* Quitte et redémarre.

**Exemple**

*INIT* (équivalent à *INIT:0*)
*INIT:2*

---

**Nom**

*!* -- Stop (arrêt forcé)

**Syntaxe**

*!* Provoque un arrêt d'urgence et stoppe l'exécution du programme.
*!!* Provoque un arrêt normal et stoppe l'exécution du programme.
*!!!* Provoque un arrêt d'urgence, stoppe l'exécution du programme et initialise l'environnement logiciel.

**Paramètres**

Aucun

**Description**

Force l'arrêt de l'exécution du programme.
Utilisé notamment avec l'AMS.

**Exemple**

*!*

---

ACTIONS

**Nom**

*PLAY* – Effectue une action

**Syntaxe**

Forme générale :

*PLAY:ACTION:<action>[:<arg1>:<arg2>:<arg3>]*

*PLAY:MWCID:<mwcid>[:<arg1>:<arg2>:<arg3>]*

**Paramètres**

*<action>* -- Nom de l'action

*<arg1 - 3>* -- Paramètre optionnel dépendant de l'action demandée

*<mwcid>* MWCommandID

**Description**

Effectue l'action désignée par <*action*>.

Les noms des actions disponibles sont enregistrés dans le fichier ACTION.AMS (/OPEN-R/APP/PC/AMS/ACTION.AMS).

Effectue l'action désignée par <*mwcid*>.

La MWCommand appelée peut être utilisée uniquement si <*mwcid*> est répertorié.

La position actuelle du robot et la position de départ de l'action à effectuer sont vérifiées.

Si elles ne sont pas identiques, AIBO fait automatiquement la transition.

L'action désignée par <*action*> ou <*mwcid*> sera faite une fois AIBO dans la bonne position.

**Paramètres**

<*angle horizontal*>:<*distance*> // Pour les commandes WALK, KICK, TOUCH, etc.
<*angle horizontal*> // Pour TURN, etc.
<*angle horizontal*>:<*angle vertical*> // Pour MOVE_HEAD, etc.
<*CIBLE*> // Pour SEARCH, etc.

L'angle est exprimé en degrés et la distance en millimétres.

Pour l'instant, seul *PINK_BALL* peut être désigné par <*CIBLE*>.

**Exemple**

*PLAY:ACTION:STAND*

---

**Nom**

*STOP* – Arrêt normal

**Syntaxe**

*STOP*

**Paramètres**

Aucun

**Description**

Arrêt normal des opérations.

Toutes les actions qui se trouvent dans la file d'attente sont annulées.

Si une action est en cours alors que *STOP* est appelé, elle est menée à son terme normalement.

**Exemple**

*STOP*

---

**Nom**

*QUIT* – Arrêt d'urgence

**Syntaxe**
*QUIT*

**Paramètres**
Aucun

**Description**
Arrêt d'urgence.
Toutes les actions qui se trouvent dans la file d'attente sont annulées.
Stoppe immédiatement toutes les opérations, même celle qui est en cours.

Remarque : Si *QUIT* est utilisé, la posture du robot se trouvera dans un état indéfini.

**Exemple**
*QUIT*

---

**Nom**
*WAIT* – Attend la fin d'une action.

**Syntaxe**
*WAIT* Attend la fin de l'action lancée juste avant.

*WAIT:<ms>* Attend <*ms*> millisecondes.

**Paramètres**
<*ms*> millisecondes [1 - 30000] Remarque : Le pas est de 32 millisecondes.

**Description**
Attend la fin de l'action démarrée juste avant.
Gèle temporairement l'exécution du programme.

*WAIT* n'a aucun effet s'il n'y a aucune opération en cours.

Si <*ms*> est spécifié, le processus est bloqué pour la durée indiquée.

**Exemple**
*WAIT*
*WAIT:1000*

---

**Nom**
*SYNC* – Synchronisation avec un système externe.

**Syntaxe**
*SYNC*

**Paramètres**
Aucun

**Description**

*SYNC* met le programme en cours d'exécution en pause et attend une entrée en provenance de l'extérieur. Le processus reprend son cours quand la chaîne "SYNC" est envoyée.

**Exemple**

*SYNC*

---

**Nom**

*NONE* -- Ne rien faire.

**Syntaxe**

*NONE*

**Paramètres**

Aucun

**Description**

Une instruction qui demande au robot de ne rien faire.
Le traitement poursuit son cours dès la ligne suivante.

**Exemple**

*NONE*

---

AFFICHAGE

**Nom**

*PRINT* – Pour afficher du texte (utile pour le débugage à distance).

**Syntaxe**

*PRINT:<format>[:<var1>:<var2>:<var3>:<var4>:<var5>]*

**Paramètres**

*<format>* Forme générale de la chaîne de caractères

%d Notation décimale
%x Notation hexadécimale

*<var1>* - *<var5>* Noms des variables incluses à la chaîne de caractères.

**Description**

Cette commande fonctionne de manière similaire à l'instruction printf() du langage C.

Si *<format>* contient des espaces ou des tabulations, délimitez *<format>* avec des guillemets ("").

**Exemple**

*PRINT:"x=%d y=%d":x:y*

*SET:age:25*
*PRINT:"TME a %d ans.":age* --> Ceci affichera « TME a 25 ans. »

---

STRUCTURES DE CONTROLE

**Nom**

*:<label>* -- Label (une sorte de marque-page dans le programme).

**Syntaxe**

*:<label>*

**Paramètres**

*<label>* Texte au choix

**Description**

Définit un label utilisable comme point de branchement pour un *GO* ou un *IF*.
Egalement utilisé pour nommer des sous-routines appelées avec *CALL*.

Un label doit obligatoirement être suivi d'une portion de programme pour avoir un quelconque intérêt. L'exécution du programme se poursuit sur la ligne juste en dessous du label. Les labels peuvent être appelés de n'importe où dans le programme.

**Exemple**

*GO:Saute*
*PRINT:"Ce texte ne s'affichera jamais."*
*:Saute*
*PRINT:"Sauté !"*

---

**Nom**

*GO* – Instruction de saut.

**Syntaxe**

*GO:<label>*

**Paramètres**

*<label>* Nom du label vers lequel on saute.

**Description**

Le programme poursuit son exécution à la ligne suivant *<label>*.
D'un point de vue purement technique, vous pouvez utiliser *GO* pour sortir d'une boucle, mais malheur à celui que je prend à programmer de cette manière :-) ON NE SORT JAMAIS D'UNE BOUCLE AVEC GO, C'EST SALE.

**Exemple**

*GO:Saute*
*PRINT:"Ce texte ne s'affichera jamais."*
*:Saute*
*PRINT:"Sauté !"*

---

**Nom**

*IF* – Pour tester si une condition est remplie ou pas.

**Syntaxe**

Syntaxe 1 :

*IF:<v1>:<op>:<v2>:THEN*
*-- THEN Block --*
*ELSE*
*-- ELSE Block --*
*ENDIF*

Syntaxe 2 :

*IF:<v1>:<op>:<v2>:CALL:<label>[:<argc>]*

Syntaxe 3 :

*IF:<v1>:<op>:<v2>:BREAK*

Syntaxe 4 :

*IF:<v1>:<op>:<v2>:<then>[:<else>]*

**Paramètres**

| | |
|---|---|
| *<v1>* | Opérande 1 |
| *<op>* | Opérateur relationnel |
| *<v2>* | Opérande 2 |
| *<label>* | Nom de la sous-routine |
| *<argc>* | Nombre de paramètres requis par la sous-routine |
| *<then>* | Point de saut si la condition est satisfaite |
| *<else>* | Point de saut si la condition n'est pas satisfaite |

Liste des opérateurs relationnels

| | |
|---|---|
| = | Egal |
| == | Est égal à |
| <> | Est différent de |
| != | Est différent de |
| < | Inférieur à |
| <= | Inférieur ou égal à |
| > | Supérieur à |
| >= | Supérieur ou égal à |
| & | ET bit-à-bit (multiplication logique) |
| \| | OU bit-à-bit (addition logique) |
| ^ | OU EXCLUSIF bit-à-bit |
| && | ET logique (Les opérandes sont considérés comme des valeurs booléennes : 0 équivaut à FALSE / les valeurs autres que 0 équivalent à TRUE) |

|| OU logique (Les opérandes sont considérés comme des valeurs booléennes : 0 équivaut à FALSE / les valeurs autres que 0 équivalent à TRUE)

**Description**

Analyse des formules de la forme <*v1*> <*op*> <*v2*> et lance des traitements en fonction des résultats.

Si c'est la syntaxe 1 qui est employée :

_ Si l'expression est TRUE on exécute ce qu'il y a dans le bloc d'instructions suivant le *THEN*,
_ Si l'expression est FALSE on exécute ce qu'il y a dans le bloc d'instructions suivant le *ELSE*.

Le block *ELSE* est facultatif. Par contre, n'oubliez jamais le *ENDIF*.

Si c'est la syntaxe 2 qui est employée :

_ Si l'expression est TRUE on exécute le *CALL* vers le <*label*> passé en paramètre. Voir la description de *CALL* pour connaître les paramètres qu'il accepte après <*label*>.

Si c'est la syntaxe 3 qui est employée :

_ Si l'expression est TRUE on exécute *BREAK*. C'est utile pour sortir des boucles (le premier que je vois sortir d'une boucle avec un *BREAK*... ;-) Pour plus de détails, se reporter à la description de *BREAK*.

Si c'est la syntaxe 4 qui est employée :

_ Si l'expression est TRUE, on saute (*GO*) à un label (désigné par <*then*>) qui fera office de bloc *THEN*,
_ Si l'expression est FALSE, on saute (*GO*) à un label (désigné par <*else*>) qui fera office de bloc *ELSE.*

Alors là, celui qui utilise cette cochonnerie de syntaxe, aïe !

**Exemples**

```
IF:x:=:0:THEN
    PRINT:"x est égal à 0"
ELSE
    PRINT:"x est différent de 0"
ENDIF
```

```
IF:x:=:0:CALL:zero
PRINT:"Ce texte s'affiche en second si x=0."
:zero
PRINT:"Ce texte s'affiche en premier si x=0."
END
```

*IF:x:<:5:BREAK*

*IF:x:<:5:vilainlabel*

---

**Nom**

*SWITCH* – Branchement multiple.

**Syntaxe**

*SWITCH:<v>*

**Paramètres**

*<v>* Nom de la variable dont la valeur va conditionner le branchement.

**Description**

La valeur de *<v>* est chargée dans une variable spéciale du système (appelée *variable de contexte*) dont certaines commandes R-Code se servent (ex. : *CASE*, *CSET*, ...) pour tester des conditions.

**Exemple**

```
SWITCH:x
CASE:1:PRINT:"x = 1"
CASE:2:PRINT:"x = 2"
CASE:2:PRINT:"(deux)"
CASE:3:PRINT:"x = 3"
CASE:ELSE:PRINT:"x ne vaut pas 1"
CASE:ELSE:PRINT:"x ne vaut pas 2"
CASE:ELSE:PRINT:"x ne vaut pas 3"
```

Dans cet exemple, les instructions *CASE* vont toutes contrôler la valeur de *x* pour la comparer au chiffre passé en argument. Si *x* est inférieur à 1 ou supérieur à 3, les trois *CASE:ELSE* de la fin sont exécutés.

---

**Nom**

*CSET* -- Branchement multiple.

**Syntaxe**

*CSET:<v1>:<op>:<v2>:<v3>*

**Paramètres**

| | |
|---|---|
| *<v1>* | Premier opérande. |
| *<op>* | Opérateur de comparaison. |
| *<v2>* | Second opérande. |
| *<v3>* | Nom de variable ou constante. |

**Description**

L'instruction *Context SET* traite des expressions de la forme *<v1>* *<op>* *<v2>* et si

le résultat est TRUE, place la valeur de *<v3>* dans la *variable de contexte (Context Value)*. Si plusieurs commandes *CSET* sont utilisées l'une après l'autre, tous les *CSET* situés après celui dont le résultat est TRUE sont ignorés.

**Exemple**

```
CSET:x:<:10:1
CSET:x:<:20:2
CSET:x:<:30:3
CSET:x:<:40:4
CSET:x:<:50:5
CASE:1:PRINT:"x < 10"
CASE:2:PRINT:"x < 20"
CASE:3:PRINT:"x < 30"
CASE:4:PRINT:"x < 40"
CASE:5:PRINT:"x < 50"
```

Remarque : *CSET:x:<:10:1* est équivalent à

```
IF:x:<:10:THEN
        SET:<variable de contexte>:1
ELSE
        IF:x:<:20:THEN
                SET:<variable de contexte>:2
        ELSE
                IF:x:<:30:THEN
                        SET:<variable de contexte>:3
                ELSE
                        IF:x:<:40:THEN
                                SET:<variable de contexte>:4
                        ELSE
                                IF:x:<:50:THEN
                                        SET:<variable de contexte>:5
                                ENDIF
                        ENDIF
                ENDIF
        ENDIF
ENDIF
IF:<variable de contexte>:=:1:THEN
        PRINT:"x < 10"
ENDIF
IF:<variable de contexte>:=:2:THEN
        PRINT:"x < 20"
ENDIF
IF:<variable de contexte>:=:3:THEN
        PRINT:"x < 30"
ENDIF
IF:<variable de contexte>:=:4:THEN
        PRINT:"x < 40"
ENDIF
IF:<variable de contexte>:=:5:THEN
        PRINT:"x < 50"
ENDIF
```

---

**Nom**

*CASE* -- Branchement multiple.

**Syntaxe**

*CASE:<constante>:<commande>*

*CASE:ELSE:<commande>*

**Paramètres**

*<constante>* Valeur constante.

*<commande>* N'importe quelle commande R-Code à l'exception de *CASE*.

**Description**

Si la *variable de contexte* est égale à *<constante>*, *<commande>* est exécuté.

L'utilisation de *CASE:ELSE* à la fin d'une série de *CASE* est possible. Si la *variable de contexte* n'est égale à aucune des constantes passées en paramètre aux *CASE*, alors *CASE:ELSE:<commande>* est exécuté.

**Exemple**

*//On met la valeur contenue dans* x *dans la* variable de contexte
*SWITCH:x*
*//Si le contenu de la* variable de contexte *vaut 1, on affiche « x=1 »*
*CASE:1:PRINT:"x = 1"*
*//Si le contenu de la* variable de contexte *vaut 2, on affiche « x=2 », puis « (deux) »*
*CASE:2:PRINT:"x = 2"*
*CASE:2:PRINT:"(deux)"*
*//Si le contenu de la* variable de contexte *vaut 3, on affiche « x=3 »*
*CASE:3:PRINT:"x = 3"*
*//Si le contenu de la* variable de contexte *est inférieur à 0 ou supérieur à 3, on affiche « x ne vaut pas 1 », puis "x ne vaut pas 2" et enfin "x ne vaut pas 3"*
*CASE:ELSE:PRINT:"x ne vaut pas 1"*
*CASE:ELSE:PRINT:"x ne vaut pas 2"*
*CASE:ELSE:PRINT:"x ne vaut pas 3"*

---

**Nom**

*FOR...NEXT* – Structure itérative (pour faire des boucles).

**Syntaxe**

*FOR:<var>:<from>:<to>[:<step>]*
*-- commandes --*
*NEXT*

**Paramètres**

*<var>* Variable incrémentée ou décrémentée à chaque tour de boucle.
*<from>* Valeur de départ.
*<to>* Valeur de fin.

*<step>* Pas d'incrémentation ou de décrémentation.

**Description**

Exécute *-- commandes --* jusqu'à *NEXT* tout en incrémentant/décrémentant *<var>* de *<from>* à *<to>* par palliers de *<step>*. A utiliser de préférence si on connaît le nombre d'itérations. Si ce n'est pas le cas, il vaut mieux se rabattre sur *WHILE*.

**Exemple**

```
//On initialise la variable i à 1 et on boucle tant que son contenu est inférieur à 11
FOR:i:1:10
    //On affiche « i=<la valeur de i> »
    PRINT:"i=%d":i
//Fin de la boucle Pour
NEXT

//On initialise la variable i à 10 et on boucle tant que son contenu est supérieur à 0
FOR:i:10:1:-1
    PRINT:"i=%d":i
NEXT
```

---

**Nom**

*WHILE...WEND* – Structure itérative (pour faire des boucles).

**Syntaxe**

```
WHILE:<v1>:<op>:<v2>
    -- commandes --
WEND
```

**Paramètres**

*<v1>* Premier opérande.
*<op>* Opérateur de comparaison.
*<v2>* Second opérande.

**Description**

Traite des expressions de la forme *<v1><op><v2>* et exécute *-- commandes –* jusqu'à *WEND* tant qu'elles sont TRUE.

**Exemple**

```
//On met la variable i à 0
SET:i:0
//Tant que la valeur de la variable i est inférieure à 10...
WHILE:i:<:10
    //...on affiche « i=<la valeur de i> »
    PRINT:"i=%d":i
//Fin de la boucle Tant que
WEND
```

---

**Nom**

*REPEAT...UNTIL* -- Structure itérative (pour faire des boucles).

**Syntaxe**

*REPEAT*
    *-- commandes --*
*UNTIL:<v1>:<op>:<v2>*

**Paramètres**

| | |
|---|---|
| *<v1>* | Premier opérande. |
| *<op>* | Opérateur de comparaison. |
| *<v2>* | Second opérande. |

**Description**

Exécute les *-- commandes –* entre *REPEAT* et *UNTIL* jusqu'à ce que l'expression *<v1><op><v2>* soit TRUE. Les *-- commandes –* sont toujours exécutées au moins une fois.

**Exemple**

```
SET:i:0
REPEAT
    PRINT:"i=%d":i
UNTIL:i:>=:10
```

---

**Nom**

*DO...LOOP* -- Structure itérative (pour faire des boucles).

**Syntaxe**

*DO[:WHILE|UNTIL:<v1>:<op>:<v2>]*
    *-- commandes --*
*LOOP[:WHILE|UNTIL:<v1>:<op>:<v2>]*

**Paramètres**

| | |
|---|---|
| *<v1>* | Premier opérande. |
| *<op>* | Opérateur de comparaison. |
| *<v2>* | Second opérande. |

**Description**

Exécute de manière cyclique les *-- commandes –* situées entre *DO* et *LOOP*. Si *WHILE* est spécifié, la boucle continue tant que la condition passée en paramètre est TRUE. Si *UNTIL* est spécifié, la boucle continue jusqu'à ce que la condition soit TRUE. Si aucune condition n'est indiquée, le bloc *DO...LOOP* devient une boucle sans fin.

**Exemple**

```
DO
    PRINT:"Boucle sans fin"
LOOP

DO
LOOP:UNTIL:Face:=:1
PRINT:"J'ai reconnu un visage !"
```

*DO*
*LOOP:WHILE:Face:=:0*
*PRINT:"J'ai reconnu un visage !"*

---

**Nom**

*BREAK* – Force la sortie d'une boucle itérative.

**Syntaxe**

*BREAK[:<niveau_de_sortie>]*

**Paramètres**

*<niveau_de_sortie>* Utile en cas de boucles imbriquées. Indique de combien de niveaux remonter. La valeur par défaut est 1. Seuls des chiffres sont acceptés en paramètres.

**Description**

Force l'arrêt de la boucle en cours. Fonctionne de la même manière avec toutes les structures itératives.

**Exemple**

*FOR:i:1:100*
*IF:i:>:10:BREAK*
*PRINT:"i=%d":i*
*NEXT*

---

**Nom**

*CALL* – Appel à une sous-routine.

**Syntaxe**

*CALL:<label>[:<argc>]*

**Paramètres**

*<label>* Nom du label de la sous-routine à lancer.
*<argc>* Nombre de paramètres passés à la sous-routine.

**Description**

Appelle la sous-routine désignée par *<label>*. On peut passer le nombre de paramètres qu'on veut à condition de les avoir mis sur la Pile avant en utilisant la commande *PUSH*. *<argc>* indique le nombre de variables placées sur la Pile à destination de la sous-routine.

Quand on appelle une sous-routine, la *variable de contexte* est *PUSH*ée sur la Pile. Elle est restaurée lorsque *RETURN* provoque la sortie de la sous-routine.

**Exemple**

*CALL:sub1*

*PUSH:10 // On place un premier argument sur la Pile...*
*PUSH:200 // ...puis un second...*

*PUSH:3000  // ...et enfin, un troisième.*
*//Là, on appelle la sous-routine nommée* sub2 *et on lui indique qu'on a 3 paramètres pour elle sur la Pile.*
*CALL:sub2:3*

---

**Nom**

*ARG* – Récupére des arguments laissés sur la Pile à destination d'une sous-routine.

**Syntaxe**

*ARG:<var>*

**Paramètres**

*<var>* Nom de la variable.

**Description**

Récupére les arguments destinés à une sous-routine dans la variable désignée par *<var>*. On doit récupérer autant de paramètres qu'on en a *PUSH*é sur la Pile. Ils sont récupérés selon une logique FIFO (First In First Out ; Premier Entré Premier Sorti). Il ne s'agit donc pas ici de FILO (First In Last Out ; Premier Entré Dernier Sorti) comme cela est le cas sur une Pile ordinaire.

**Exemple**

```
//On prépare l'appel à sub
//en mettant trois paramètres sur la Pile
PUSH:10
PUSH:200
PUSH:3000
CALL:sub
...

//Et voici la sous-routine
:sub
ARG:arg1  // arg1 <- 10
ARG:arg2  // arg2 <- 200
ARG:arg3  // arg3 <- 3000
...
// arg1, 2 et 3 sont considérés comme des variables locales
//donc utilisables uniquement dans la sous-routine.
//Il n'y a pas besoin d'utiliser la commande LOCAL.
```

---

**Nom**

*RETURN* – Provoque la sortie d'une sous-routine.

**Syntaxe**

*RETURN[:<valeur_retournee>]*

**Paramètres**

*<valeur_retournee>* Valeur retournée par la sous-routine.

**Description**

Provoque la sortie d'une sous-routine. <*valeur_retournee* > peut-être spécifié. Par exemple, c'est utile pour savoir si l'exécution de la sous-routine s'est bien déroulée (on retourne 0 si tout est OK ou 1 s'il y a eu un souci). La valeur retournée est récupérée grâce à *POP*.

**Exemple**

*//C'est ici qu'on appelle notre sous-routine nommée* sub
*CALL:sub*
*POP:rc //On récupére 123 dans la variable* rc*. Valable uniquement si la sous-routine retourne quelque chose.*
*...*

*//La sous-routine*
*:sub*
*...*
*RETURN:123*

---

**Nom**

*RET* – Stoppe une sous-routine en plaçant une valeur de retour dans la *variable de contexte.*

**Syntaxe**

*RET:<contexte>*

**Paramètres**

*<contexte>* Valeur à placer dans la *variable de contexte.*

**Description**

Provoque la sortie d'une sous-routine et place le contenu de *<contexte>* dans la *variable de contexte.* Si *<contexte>* vaut 0, alors la *variable de contexte* reprend la valeur qu'elle avait au moment de l'appel à la sous-routine.

**Exemple**

*CALL:sub*
*CASE:1:PRINT:"La variable de contexte vaut 1."*
*CASE:2:PRINT:"La variable de contexte vaut 2."*
*CASE:3:PRINT:"La variable de contexte vaut 3."*
*...*

*:sub*
*...*
*//On quitte la sous-routine et on place*
*//la valeur 1 dans la* variable de contexte*.*
*RET:1*
*...*
*//On quitte la sous-routine et on place*
*//la valeur 2 dans la* variable de contexte*.*
*RET:2*
*...*

*//On quitte la sous-routine et on place*
*//la valeur 3 dans la* variable de contexte*.*
*RET:3*

---

**Nom**

*ONCALL* – Commande de gestion d'interruptions.

**Syntaxe**

*ONCALL:<v1>:<op>:<v2>:<label>[:<resume_type>:<resume_label>]*

*ONCALL:<-n>*

**Paramètres**

| | |
|---|---|
| *<v1>* | Premier opérande. |
| *<op>* | Opérayeur de comparaison. |
| *<v2>* | Second opérande. |
| *<label>* | Label qui est appelé lorsque *<v1><op><v2>* est TRUE. |
| *<resume_type>* | Détermine le comportement à adopter à la sortie de la sous-routine. |
| *<resume_label>* | Indique l'endroit du programme auquel on saute à la sortie de la sous-routine. |
| *<-n>* | Permet d'annuler les *n* derniers *ONCALL*. |

**Description**

La sous-routine désignée par *<label>* est appelée (*CALL*ed) quand *<v1><op><v2>* est TRUE. Le principe est le même que pour *IF*.

Ce qui se passe à la sortie de la sous-routine est fonction de la valeur de *<resume_type>* :

0 : Retourne à la portion du programme qui a généré l'interruption à l'origine de l'exécution de la sous-routine.
1 : Retourne à la portion du programme qui a généré l'interruption à l'origine de l'exécution de la sous-routine, puis saute à *<resume_label>*.
2 : Retourne à la ligne où figure la commande *ONCALL* définissant l'interruption.
3 : Retourne à la ligne où figure la commande *ONCALL* définissant l'interruption, puis saute à *<resume_label>*.
4 : Retourne au début du programme (la Pile est entièrement vidée).
5 : Retourne au début du programme (la Pile est entièrement vidée), puis saute à *<resume_label>*.

Dans les cas 2 à 5, tous les *ONCALL* définis après celui étant à l'origine de l'exécution de la sous-routine sont annulés. Dans les cas 3 et 5, tous les *ONCALL* définis au même niveau que le point de retour sont préservés.

*<v1><op><v2>* est vérifié à chaque ligne du programme pour savoir s'il l'expression est TRUE ou FALSE au fur et à mesure de l'exécution.

Un *ONCALL* effectué dans une sous-routine est effacé lorsqu'on sort de ladite sous-routine.

Si le paramètre <-*n*> est spécifié, les *n* derniers *ONCALL* sont annulés, ce qui fait que si les interruptions auxquelles ils font références se produisent, il n'y aura pas de saut aux labels désignés. <-*n*> doit être un nombre négatif.

**Exemple**

*//Ici, on saute au label* 9000 *si la variable* RFLeg_ON *passe à 1*
*ONCALL:RFLeg_ON:=:1:9000*

---

**Nom**

*RESUME* – Sort d'une sous-routine exécutée suite à un *ONCALL*.

**Syntaxe**

*RESUME*

**Paramètres**

Aucun

**Description**

Sort d'une sous-routine exécutée suite à un *ONCALL*. *RESUME* est à *ONCALL* ce que *RETURN* est à *CALL*. *RETURN* ne doit pas être utilisé pour sortir d'une sous-routine liée à une interruption. Le point de retour dépend de la valeur de *<resume_type>* passé en paramètre à *ONCALL*.

**Exemple**

*RESUME*

---

VARIABLES

**Nom**

*GLOBAL* – Déclare une variable globale (accessible partout dans le programme).

**Syntaxe**

*GLOBAL:<var>[:<valeur_initiale>]*

**Paramètres**

| | |
|---|---|
| *<var>* | Nom de la variable. |
| *<valeur_initiale>* | Valeur initiale de la variable. |

**Description**

Crée et initialise la variable globale désignée par <var>. Si *<valeur_initiale>* est spécifié, la variable est initialisée à cette valeur. Si une variable est créée sans que *GLOBAL* ou *LOCAL* soit indiqué, elle est considérée par défaut comme étant globale.

**Exemple**

*GLOBAL:x:0*

---

**Nom**

*LOCAL* -- Déclare une variable locale (accessible uniquement dans la sous-routine où elle est déclarée).

**Syntaxe**

*LOCAL:<var>[:<valeur_initiale>]*

**Paramètres**

| | |
|---|---|
| *<var>* | Nom de la variable. |
| *<valeur_initiale>* | Valeur initiale de la variable. |

**Description**

Crée et initialise la variable locale désignée par <var>. Si *<valeur_initiale>* est spécifié, la variable est initialisée à cette valeur. L'espace mémoire occupé par une variable locale est libéré dès que celle-ci est détruite (au moment où on sort de la sous-routine dans laquelle elle a été déclarée par exemple).

**Exemple**

*LOCAL:y:0*

---

**Nom**

*LET* – Attribue une valeur à une variable.

**Syntaxe**

*LET:<var>:<valeur>*

**Paramètres**

| | |
|---|---|
| *<var>* | Nom de la variable. |
| *<valeur>* | Valeur (peut-être une variable ou une constante). |

**Description**

Attribue une valeur à une variable. Ne pas confondre avec *SET* qui ne sert qu'à traiter des variables du système.

**Exemple**

*LET:x:1*

---

**Nom**

*SET* – Attribue une valeur à une variable système.

**Syntaxe**

*SET:<var>:<valeur>*

**Paramètres**

| | |
|---|---|
| *<var>* | Nom de la variable. |

*<valeur>* Valeur (peut-être une variable ou une constante).

**Description**

Attribue une valeur à une variable système. Ne pas confondre avec *LET*.

*Power* Sert d'interrupteur de mise en marche.
** 09/03/2001 ***
Si *SET:Power:1* est exécuté alors qu'AIBO s'est mis en mode sécurité suite à un choc, il se rallume. Pour des raisons de sécurité, la varieble *Power* n'est désormais plus utilisée pour allumer ou éteindre le robot. Elle reste cependant disponible, de sorte que vous n'avez aucune modification à apporter à vos programmes.

*Head_ON* Il faut utiliser SET pour remettre ces variables à 0
*Head_LONG* si leur état a été changé par les capteurs.
*BackF_ON*
*BackM_ON*
*BackR_ON*
*BackFR_LONG*
*BackF_Jaw_LONG*
*Back_Pat*
*BackR_Hit*
*Jaw_ON*
*RFLeg_ON*
*RFLeg_OFF*
*LFLeg_ON*
*LFLeg_OFF*
*RRLeg_ON*
*RRLeg_OFF*
*LRLeg_ON*
*LRLeg_OFF*

**Exemple**

*SET:Power:1*

---

**Nom**

*GET* – Récupére et affiche la valeur d'une variable.

**Syntaxe**

*GET:<var>*

**Paramètres**

*<var>* Nom de la variable.

**Description**

Récupére et affiche la valeur de la variable *<var>* sur la console Wi-Fi. Utile pour faire la chasse aux bugs.

**Exemple**

*GET:x*

---

**Nom**

*CLR* – Réinitialise les variables indiquant l'état des capteurs.

**Syntaxe**

*CLR:SENSORS*

**Paramètres**

Aucun

**Description**

Met à 0 la valeur des variables indiquant l'état des capteurs.

**Exemple**

*CLR:SENSORS*

---

**Nom**

*VSAVE* – Sauvegarde la valeur d'une variable.

**Syntaxe**

*VSAVE:<var>*

**Paramètres**

*<var>* Nom de la variable.

**Description**

Sauvegarde la valeur d'une variable dans le fichier /OPEN-R/APP/PC/AMS/<nom de la variable>.SAV sur le Memory Stick.

ATTENTION !

Le système de fichiers utilisé ne gére pas les noms de variables de plus de 8 caractères + extension (format 8+3, comme en MS-DOS).

**Exemple**

*VSAVE:x*

---

**Nom**

*VLOAD* – Charge la valeur d'une variable.

**Syntaxe**

*VLOAD:<var>*

**Paramètres**

*<var>* Nom de la variable.

**Description**

Lit la valeur d'une variable dans le fichier /OPEN-R/APP/PC/AMS/<nom de la variable>.SAV sur le Memory Stick.

ATTENTION !
Le système de fichiers utilisé ne gére pas les noms de variables de plus de 8 caractères + extension (format 8+3, comme en MS-DOS).

**Exemple**
*VLOAD:x*

---

**Nom**
*VDUMP* – Affiche la valeur d'une variable.

**Syntaxe**
*VDUMP:<var>*

**Paramètres**
*<var>* Nom de la variable.

**Description**
Affiche la valeur de la variable sur la console Wi-Fi de la manière suivante :

*<nom de la variable> = <valeur de la variable>*

Cette commande est utile pour faire du débugage via le réseau sans-fil.

**Exemple**
*VDUMP:x*

---

OPERATEURS

Ils effectuent des opérations arithmétiques dont les résultats sont stockés dans *<var>*.

*<var>* Nom de la variable
*<valeur>* Valeur (nom d'une variable ou valeur constante)

| | | |
|---|---|---|
| *ADD:<var>:<valeur>* | <var> ← <var> + <valeur> | Addition |
| SUB:<var>:<valeur> | <var> ← <var> - <valeur> | Soustraction |
| *MUL:<var>:<valeur>* | <var> ← <var> * <valeur> | Multiplication |
| *DIV:<var>:<valeur>* | <var> ← <var> / <valeur> | Division |
| *MOD:<var>:<valeur>* | <var> ← <var> % <valeur> | Reste de la division |
| *AND:<var>:<valeur>* | <var> ← <var> & <valeur> | ET bit-à-bit |
| *IOR:<var>:<valeur>* | <var> ← <var> \| <valeur> | OU bit-à-bit |
| *XOR:<var>:<valeur>* | <var> ← <var> ^ <valeur> | OU exclusif bit-à-bit |
| *NOT:<var>:<valeur >* | <var> ← ~ <valeur> | NON logique |
| *LAND:<var>:<valeur >* | <var> ← <var> AND <valeur> | ET logique |
| *LIOR:<var>:<valeur >* | <var> ← <var> OR <valeur> | OU logique |

*LNOT:<var>:<valeur >* <var> ← NOT <valeur> NON logique

*RND:<var>:<de>:<a>* *<var>* ← nombre aléatoire compris entre <*de*> et <*a*>

Le nombre servant de base à la création des chiffres aléatoires est indiqué par *SET:Seed:<nombre de base>*.

En plus des opérations ci-dessus, voici la liste des opérateurs agissant sur la Pile : *POP* retire un élément de la Pile et *PUSH* y place le résultat.

| | | |
|---|---|---|
| *ADD* | <push> ← <pop2> + <pop1> | Addition |
| *SUB* | <push> ← <pop2> - <pop1> | Soustraction |
| *MUL* | <push> ← <pop2> * <pop1> | Multiplication |
| *DIV* | <push> ← <pop2> / <pop1> | Division |
| *MOD* | <push> ← <pop2> % <pop1> | Reste de la division |
| *AND* | <push> ← <pop2> & <pop1> | ET bit-à-bit |
| *IOR* | <push> ← <pop2> \| <pop1> | OU bit-à-bit |
| *XOR* | <push> ← <pop2> ^ <pop1> | OU exclusif bit-à-bit |
| *NOT* | <push> ← ~ <pop1> | NON logique |
| *LAND* | <push> ← <pop2> AND <pop1> | ET logique |
| *LIOR* | <push> ← <pop2> OR <pop1> | OU logique |
| *LNOT* | <push> ← NOT <pop1> | NON logique |
| *EQ* | <push> ← <pop2> == <pop1> | Est égal à |
| *NE* | <push> ← <pop2> <> <pop1> | N'est pas égal à |
| *LT* | <push> ← <pop2> < <pop1> | Est inférieur à |
| *LE* | <push> ← <pop2> <= <pop1> | Est inférieur ou égal à |
| *GT* | <push> ← <pop2> > <pop1> | Supérieur à |
| *GE* | <push> ← <pop2> >= <pop1> | Supérieur ou égal à |

*RND:<de>:<a>* <push> ← nombre aléatoire compris entre <de> et <a>.
*RND:<a>* <push> ← nombre aléatoire compris entre 0 et <a>.

Remarque : Les éléments sont extraits de la Pile (*POP*ée) dans l'ordre de *<pop1>* et *<pop2>*.

*PUSH:x*
*PUSH:y*
*SUB*
*POP:z*
*//Ici, la valeur de* z *devient* x - y *(et pas* y – x*).*

MANIPULATION DE LA PILE

*PUSH:<var>* La valeur de la variable ou de la constante est mise (*PUSH*ée) sur la Pile.

*POP[:<var>]* Met une valeur de la Pile dans une variable. Si *<var>* n'est pas indiqué, la valeur récupérée n'est pas utilisée.

*DUP* Copie l'élément situé en haut de la Pile et le place sur la Pile.

*JT:<label>* Prend une valeur sur la Pile et saute à *<label>* si elle est TRUE.

*JF:<label>* Prend une valeur sur la Pile et saute à *<label>* si elle est FALSE.

Remarque : Les opérations effectuées sur la Pile sont soumises aux restrictions induites par l'utilisation de sous-routines et au principe de portée.

## 2.6 – Les opérateurs logiques simples

= Egal
== Est égal à
<> Est différent de
!= Est différent de
< Inférieur à
<= Inférieur ou égal à
> Supérieur à
>= Supérieur ou égal à
& ET bit-à-bit (multiplication logique)
| OU bit-à-bit (addition logique)
^ OU EXCLUSIF bit-à-bit
&& ET logique
(Les opérandes sont considérés comme des valeurs booléennes
0 équivaut à FALSE / les valeurs autres que 0 équivalent à TRUE)
|| OU logique
(Les opérandes sont considérés comme des valeurs booléennes
0 équivaut à FALSE / les valeurs autres que 0 équivalent à TRUE)

## 2.7 – Les variables système

*AiboId* AIBO ID (0-255)
Est à 0 quand la connexion est établie par Wi-Fi.
Quand la connexion est établie par Wi-Fi, c'est le bit de poids faible de l'adresse IP.

| | |
|---|---|
| *AiboType* | Modèle |
| 7 | ERS-7 |

| | |
|---|---|
| *Year* | Année (2000 ou plus) |
| *Month* | Mois (1-12) |
| *Day* | Jour (1-31) |
| *Hour* | Heure (0-23) |
| *Min* | Minutes (0-59) |
| *Sec* | Secondes (0-59) *Précision : 2 secondes |
| *Dow* | Jour de la semaine (Sun(0), Mon(1), ..., Sat(6)) |

*Seed* Base pour la génération de nombre aléatoire (1 par défaut)

*Power* Alimentation (0/1) :
0 ARRET
1 MARCHE
N'a aucune utilité pour les versions 1.1 et ultérieures.
Voir la description de la commande *SET* pour plus de détails.

*Status* Etat :
0 Démarrage normal
1 Démarrage suite à une chute
Si AIBO se relève d'une chute, le programme reprend depuis le début.
A ce moment-là, la valeur est à 1.

*Context* *Variable de contexte*

*Wait* Nombre d'opérations dont l'exécution est prévue.

*Clock* Horloge (incrémentée de 1 toutes les 32 ms)

*Brightness* Luminosité ambiante (0-255)
*Face* Indicateur de détection de visage (0/1)

*Pink_Ball* Balle rose (0/1) *
*Pink_Ball_H* Angle horizontal par rapport à la balle rose [degrés]
*Pink_Ball_V* Angle vertical par rapport à la balle rose [degrés]
*Pink_Ball_D* Distance par rapport à la balle rose [mm]

*AIBONE* AIBONE (l'OS rose et blanc d'AIBO) (0/1) *
*AIBONE_H* Angle horizontal par rapport à l'AIBONE [degrés]
*AIBONE_V* Angle vertical par rapport à l'AIBONE [degrés]
*AIBONE_D* Distance par rapport à l'AIBONE [mm]

*AU_Voice* Indicateur de reconnaissance vocale (0/1) *
*AU_Voice_ID* ID de la voix (1-53) voir [section 2.9]
*AU_AiboSound* Indicateur de détection d'un AiboSound (0/1) *
*AU_AiboSound_ID* ID de l'AiboSound (1-35) voir [AiboSound ID List]
*AU_AiboTone* Indicateur de détection d'un AiboTone (0/1) *
*AU_AiboTone_ID* ID de l'AiboTone (1-68) voir [AiboTone ID List]

*Temp_Hi* Température à laquelle le robot s'arrête (0/1)
Par sécurité, un arrêt forcé est effectué au bout de 20 secondes si la température est trop élevée.

*Head_Tilt* Tête : angle vertical 1 (haut-bas) [degrés]
*Head_Tilt_2* Tête : angle vertical 2 (haut-bas) [degrés]
*Head_Pan* Tête : angle horizontal (gauche-droite) [degrés]
*Mouth* Gueule : Indice d'ouverture [degrés]

*LFLeg_1* Jambe gauche avant J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*LFLeg_2* Jambe gauche avant J1 (articulation de la hanche)

Angle indiquant le degré d'inclinaison vers la droite ou la gauche [degrés]

*LFLeg_3* Jambe gauche avant J2 (articulation du genoux)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*LRLeg_1* Jambe gauche arrière J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*LRLeg_2* Jambe gauche arrière J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers la droite ou la gauche [degrés]

*LRLeg_3* Jambe gauche arrière J2 (articulation du genoux)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*RFLeg_1* Jambe droite avant J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*RFLeg_2* Jambe droite avant J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers la droite ou la gauche [degrés]

*RFLeg_3* Jambe droite avant J2 (articulation du genoux)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*RRLeg_1* Jambe droite arrière J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*RRLeg_2* Jambe droite arrière J1 (articulation de la hanche)
Angle indiquant le degré d'inclinaison vers la droite ou la gauche [degrés]

*RRLeg_3* Jambe droite arrière J2 (articulation du genoux)
Angle indiquant le degré d'inclinaison vers l'avant ou l'arrière [degrés]

*Tail_Pan* Queue : angle horizontal (gauche/droite) [degrés]
*Tail_Tilt* Queue : angle vertical (haut/bas) [degrés]

*Batt_Rest* Indicateur de charge de la batterie [%]
*Batt_Temp* Température de la batterie [C]

*Distance_Cliff* Distance par rapport au vide [mm]
*Distance* Distance par rapport à un obstacle [mm]

*Head_ON* Capteur de tête touché (0/1) *
*Head_LONG* Capteur de tête touché pendant 3 secondes ou plus (0/1) *

*BackF_ON* Premier capteur du dos touché (0/1) *
*BackM_ON* Capteur du milieu du dos touché (0/1) *
*BackR_ON* Capteur de l'arrière touché (0/1) *
*BackFR_LONG* Premier capteur et capteur de l'arrière touchés durant 3 secondes ou plus (0/1) *
*BackF_Jaw_LONG* Premier capteur et capteur de la gueule touchés durant 3 secondes ou plus (0/1) *
*Back_Pat* Les capteurs du dos ont été caressés (0/1) *

| | | |
|---|---|---|
| *BackR_Hit* | Le capteur de l'arrière a été frappé (0/1) | * |
| *Jaw_ON* | Le capteur de la gueule a été touché (0/1) | * |
| *RFLeg_ON* | Le capteur de la patte avant-droit est enfoncé (0/1) | * |
| *RFLeg_OFF* | Le capteur de la patte avant-droit est relâché (0/1) | * |
| *LFLeg_ON* | Le capteur de la patte avant-gauche est enfoncé (0/1) | * |
| *LFLeg_OFF* | Le capteur de la patte avant-gauche est relâché (0/1) | * |
| *RRLeg_ON* | Le capteur de la patte arrière-droit est enfoncé (0/1) | * |
| *RRLeg_OFF* | Le capteur de la patte arrière-droit est relâché (0/1) | * |
| *LRLeg_ON* | Le capteur de la patte arrière-gauche est enfoncé (0/1) | * |
| *LRLeg_OFF* | Le capteur de la patte arrière-gauche est relâché (0/1) | * |

* Ces valeurs ne seront pas remises à 0 une fois modifiées par les capteurs. Vous devez les remettre à 0 en utilisant *SET*.

## 2.8 – Les actions connues par AIBO

| | |
|---|---|
| *SIT* | Assis |
| *STAND* | Debout |
| *LIE* | Couché (1) |
| *WALK* | Marche |
| *STOP_WALK* | Arrête de marcher |
| *TURN* | Tourne |
| *KICK* | Tape |
| *TOUCH* | Touche |
| *MOVE_HEAD* | Regardesdans la direction indiquée |
| *TRACK_HEAD* | Suis un objet |
| *SEARCH* | Cherche un objet |
| *SEARCH.HEAD.NORMAL* | Cherche un objet situé dans ton champ de vision actuel |
| *SEARCH.HEAD.SLOW* | Cherche attentivement un objet situé dans ton champ de vision actuel |
| *SEARCH.HEAD.NORMALCENT* | Regarde vers l'avant et cherches |
| *SEARCH.HEAD.SLOWCENT* | Regarde vers l'avant et cherches attentivement |
| *SEARCH.HEAD.LOWCENT* | Regarde vers le bas et cherches |
| *PALONE.AUTO.EAR* | Bouge les deux oreilles |
| *PALONE.AUTO.EARSTOP* | Arrête de bouger les oreilles |
| *PALONE.AUTO.TAILV* | Remue la queue de haut en bas |
| *PALONE.AUTO.TAILH* | Remue la queue de gauche à droite |
| *PALONE.AUTO.TAILROT* | Fais tourner ta queue |
| *PALONE.AUTO.TAILD* | Baisse ta queue |
| *PALONE.AUTO.TAILSTOP* | Arrête de remuer la queue |
| *MOVE.HEAD.FAST* | Regarde rapidement dans la direction indiquée |
| *MOVE.HEAD.NORMAL* | Regarde dans la direction indiquée |
| *MOVE.HEAD.SLOW* | Regarde attentivement dans la direction indiquée |
| *MOVE.TURN.NORMAL* | Tourne |
| *MOVE.TURN.SLOW* | Tourne lentement |
| *MOVE.MOVE.NORMAL* | Marche |
| *MOVE.MOVE.SLOW* | Marche lentement |
| *CHGPOS.WALK.NORMAL* | Mets-toi en position de marche |
| *CONTACT.RIGHT.TOUCH* | Assieds-toi et fais mine de toucher quelque chose |

avec la patte droite

| | |
|---|---|
| *CONTACT.RIGHT.TOUCH2* | Allonge-toi et fais mine de toucher quelque chose avec la patte droite |
| *CONTACT.FRONT.HEAD* | Donne un coup de tête dans la balle |
| *CHGPOS.STATR.NORMAL* | Couché (2) |
| *SMESS.NOTICE.NOTICE1* | Indicateur de mode Debug 1 |
| *SMESS.NOTICE.NOTICE2* | Indicateur de mode Debug 2 |
| *SMESS.ERROR.ERROR1* | Indicateur d'erreur de mode Debug 1 |
| *SMESS.ERROR.ERROR2* | Indicateur d'erreur de mode Debug 2 |
| *SMESS.MODE.CLEAR* | Effacer les messages retournés par le mode Debug |
| *CLIFF_DETECT_ON* | Activation de la détection de trous dans le sol (activé par défaut) |
| *CLIFF_DETECT_OFF* | Désactivation de la détection de trous dans le sol |

## 2.9 – Les expressions reconnues par AIBO

| ID | Mot |
|---|---|
| 1 | AIBO (AIBO) |
| 2 | Comment t'appelles-tu ? (What's your name ?) |
| 3 | Dis bonjour (Say hello) |
| 4 | Donne la patte (Shake paw) |
| 5 | Salut ! (Morning) |
| 6 | Bonjour (Hello) |
| 7 | Bonne nuit (Good night) |
| 8 | A+ (See you) |
| 9 | Comment ça va ? (How are you ?) |
| 10 | Hé AIBO ! (Hey AIBO) |
| 11 | Merci (Thanks) |
| 12 | Désolé (Sorry) |
| 13 | Bravo ! (Cheer up) <--- Ici, j'ai un doute sur la traduction |
| 14 | Banzai (Banzai) |
| 15 | C'est bien (That's right) |
| 16 | C'est mal (That's wrong) |
| 17 | Bon chien (Good AIBO) |
| 18 | Ne fais pas ça (Don't do that) |
| 19 | Jouons ! (Let's play !) |
| 20 | Chante une chanson (Sing a song) |
| 21 | Danse (Dance) |
| 22 | Montre l'heure (Show time) |
| 23 | Prend la pose (Pose for me) |
| 24 | Vas jouer (Clown around) |
| 25 | Fini de jouer (Show off) |
| 26 | Dis un message (Say message) |
| 27 | Je vais te dire un secret (Let's be secret) |
| 28 | Sésame ouvre-toi (Open sesame) |
| 29 | Happy day (Happy day) |
| 30 | Debout (Stand up) |
| 31 | Couché (Lie down) |
| 32 | Assis (Sit down) |
| 33 | Tourne à droite (Turn right) |
| 34 | Tourne à gauche (Turn left) |

| | |
|---|---|
| 35 | Marche vers l'avant (Go forward) |
| 36 | Marche vers l'arrière (Go backward) |
| 37 | Droit devant toi (Go ahead) |
| 38 | Stop (Stop) |
| 39 | Plus vite (Faster) |
| 40 | Ralentis (Slow down) |
| 41 | Balle rose (Pink ball) |
| 42 | Tape avec la patte droite (Right leg kick) |
| 43 | Touche avec la patte droite (Right leg touch) |
| 44 | Tape avec la patte gauche (Left leg kick) |
| 45 | Touche avec la patte gauche (Left leg touch) |
| 46 | Prêt, feu, partez ! (Ready set go) |
| 47 | Tu as gagné (You won) |
| 48 | Tu as perdu (You lost) <-- C'est pas plutôt YOU LOOSE ? |
| 49 | Action un (Action one) |
| 50 | Action deux (Action two) |
| 51 | Action trois (Action three) |
| 52 | Action quatre (Action four) |
| 53 | Action cinq (Action five) |

## 2.10 – Chasser les bugs

Une bonne façon de créer des programmes en R-Code est de le faire en utilisant la console Wi-Fi. Cela permet de tester et de débuguer ses programmes sans avoir à les réécrire sans cesse sur le Memory Stick. Cela vous permet en outre de voir les messages retournés par le programme.

Pour créer un programme via la console Wi-Fi, connectez-vous comme indiqué à la section 2.2.3, puis saisissez-le dans un éditeur de texte. Une fois que cela est fait, tapez *EDIT* dans la console. Faites un copier/coller du texte directement vers la ligne de commandes, puis entrez *END*. Votre nouveau programme est ainsi copié dans la mémoire d'AIBO. Il vous suffit de taper *RUN* pour le voir tourner.

R-Code met deux commandes à votre disposition pour envoyer des messages à la console :

1 – *VDUMP* : Cette commande affiche la valeur d'une variable à l'écran. La syntaxe est *VDUMP:<nom de la variable>*. Cela va afficher <nom de la variable> = <valeur de la variable>,

2 – *PRINT* : Cette commande fonctionne de manière similaire à l'instruction printf() du langage C. La syntaxe est *PRINT:<format>:<var1>:...:<var5>*. *<format>* est la forme générale de la chaîne de caractères. Si *<format>* contient des espaces ou des tabulations, délimitez *<format>* avec des guillemets (""). Exemple :

*PRINT:"x=%d y=%d":x:y*

*SET:age:25*
*PRINT:"TME a %d ans.":age* --> Ceci affichera « TME a 25 ans. »

## 3 – Exemples

Voici les deux programmes d'exemple « Greeting » et « Maze » commentés fournis par SONY dans le SDK pour R-Code.

### 3.1 – Le programme d'exemple « Greeting.R »

Ce programme utilise la reconnaissance vocale d'AIBO et lui fait faire un mouvement de salutation.

```
//————————————————————- // GREETING //————————————————————-
:START
// On appelle la sous-routine d'initialisation 1001
CALL:1001
//Un fois revenu de la sous-routine 1001, on entame une boucle (DO..LOOP)
DO
	//On attend 1 milliseconde
	WAIT:1
	//Si une voix a été reconnue...
	IF:AU_Voice:=:1:THEN
		WAIT:1
		//...on cherche à savoir quelle commande a été prononcée
		SWITCH:AU_Voice_ID
		//Si c'est « AIBO », on saute en 1003
		CASE:1:CALL:1003
		//Si c'est « HELLO », on saute en 1005
		CASE:6:CALL:1005
		//Si ce n'est ni « AIBO », ni « HELLO », on saute en 1007
		CASE:ELSE:CALL:1007
		CALL:1001
	ENDIF
	//On attend 1 seconde
	WAIT:1000
//Fin de la boucle (DO...LOOP)
LOOP
// RESET
:1001
//AIBO se lève
PLAY:ACTION:STAND
//On attend qu'AIBO ait fini de se lever (c'est très important)
WAIT
//On initialise la variable indiquant qu'une voix à été reconnue à 0 (c'est à dire, FALSE)
SET:AU_Voice:0
RETURN
//C'est la commande « AIBO » qui a été prononcée
:1003
//Le robot fait quelques pas
PLAY:ACTION:WALK:0:100
//On attend qu'AIBO ait fini de marcher
WAIT
```

```
//On fait se mouvoir la tête
PLAY:ACTION:MOVE_HEAD:0:-50
WAIT
PLAY:ACTION:MOVE_HEAD:0:0
WAIT
RETURN
//C'est la commande «HELLO» qui a été prononcée
:1005
//Equivalent à L_PHUMAN.M_GREET.S_NORMAL
//Le robot joue un son indiquant qu'il nous remercie
PLAY:MWCID:2750
WAIT
RETURN
//Le robot n'est pas content, car la commande n'a pas été comprise
:1007
//Equivalent à L_EDSP.M_ANGRY.S_SOUNDLED
// Le robot joue un son et une animation IllumFace indiquant qu'il n'est pas content
PLAY:MWCID:2471
WAIT
RETURN
```

### 3.2 – Le programme d'exemple « Maze.R »

*Le but de ce programme est de faire sortir AIBO d'un labyrinthe.*

```
//————————————————— // MAZE //—————————————————
:1000
//Fais se lever AIBO
PLAY:ACTION:STAND
//On attend la fin de l'action en cours, en l'occurence, on attend qu'AIBO ait fini de se lever
WAIT
//On désactive la détection des trous
PLAY:ACTION:CLIFF_DETECT_OFF
WAIT
//Début de la boucle DO
DO
    //On centre la tête du robot
    PLAY:ACTION:MOVE_HEAD:0:0
    WAIT
    //Le robot marche tout droit sur 10 mètres
    PLAY:ACTION:WALK:0:10000
    //Début de la boucle FOR
    FOR:t:1:1000
        //Si AIBO détecte un mur à moins de 300 millimètres ou que t vaut 1000, on quitte la boucle
        IF:Distance:<:300:BREAK
        WAIT:1
    //Fin de la boucle FOR
    NEXT
    //Le robot arrête de marcher
    PLAY:ACTION:STOP_WALK
```

```
	WAIT
	//La tête s'oriente d'un côté...
	PLAY:ACTION:MOVE_HEAD:90:0
	WAIT
	//...puis de l'autre. Vu qu'on va procéder à une mesure de distance en même temps que la tête tourne, on n'a pas besoin d'attendre la fin de l'opération avant de passer à la suite. Ceci explique l'abscence de WAIT
	PLAY:ACTION:MOVE_HEAD:-90:0
	SET:dd:0
	//Début de la boucle WHILE. Elle se poursuit jusqu'à ce que la dernière action soit terminée
	WHILE:Wait:>:0
		//On place la distance relevée dans la variable d
		SET:d:Distance
		//On place l'angle d'inclinaison de la tête relevé dans la variable p
		SET:p:Head_Pan
		IF:d:>:dd:THEN
			//La variable dd contient la plus grande distance relevée par rapport à un mur
			SET:dd:d
			//La variable pan contient l'angle de la tête par rapport à cette distance
			SET:pan:p
		ENDIF
	//Fin de la boucle WHILE
	WEND
	VDUMP:dd
	VDUMP:pan
	//Si le contenu de dd est inférieur à 300, cela signifie qu'il n'y a nulle part où aller
	IF:dd:<:300:THEN
		//AIBO fait donc un virage à 180 degrés
		PLAY:ACTION:TURN:180
		WAIT
	ENDIF
	//Si une issue est trouvée, alors on positionne la tête du robot dans sa direction...
	PLAY:ACTION:MOVE_HEAD:pan:0
	//...puis le reste du corps
	PLAY:ACTION:TURN:pan
	WAIT
//Fin de la boucle LOOP
LOOP
```

# 4 – Création et utilisation de fichiers de ressources

Bien que l'environnement R-Code de base inclue un grand nombre de séquences de mouvements, animations IllumFace et sons, vous avez la possibilité de créer les vôtres et de les utiliser dans vos programmes. Par exemple, vous pouvez concevoir vos propres pas de danse en utilisant un éditeur de mouvements et les utiliser ensuite grâce à la commande *PLAY*.

Tous les mouvements, animations IllumFace et sons que vous utilisez pour faire danser ou bouger AIBO sont stockés dans ce que l'on appelle des fichiers de ressources. Cette section explique comment les créer et les exploiter.

Voici en condensé la marche à suivre pour créer un fichier de ressources :

1. Générer le fichier. Son type dépend des données qu'il va acccueillir (les séquences de mouvements sont enregistrées dans des fichiers portant l'extension MTN, les animations IllumFace sont dans des fichiers LED et les sons peuvent être au format WAV ou MID),

2. Convertir le fichier au format ODA (Open Document Architecture ; C'est une norme de représentation de la structure de documents bureautiques),

3. Mettre à jour le fichier MWC.CFG en lui ajoutant une liste des nouveaux contenus,

4. Effectuer un traitement spécial sur MWC.CFG de manière à obtenir le fichier ERS-7.MWC,

5. Copier le fichier ODA et ERS-7.MWC sur le Memory Stick. Il ne reste plus qu'à utiliser vos nouvelles créations avec la commande *PLAY*.

Il existe plusieurs outils qui permettent de créer des fichiers de ressources ; Nous allons nous focaliser sur ceux qui sont fournis par SONY.

## 4.1 – Créer des fichiers de ressources

Il existe trois types de fichiers de ressources : Les séquences de mouvements (*.mtn), les sons (*.wav ou *.mid) et les animations IllumFace (*.led).

### 4.1.1 – Créer des fichiers de mouvements

Ces fichiers décrivent des séquences de mouvements et portent l'extension *.mtn. Ces séquences doivent être conçues par le programmeur à l'aide d'un éditeur de mouvements. Il existe plusieurs logiciels de ce type pour AIBO, comme par exemple « AIBO Master Studio » (commercialisé par SONY) ou Skitter (qui peut être téléchargé gratuitement sur le site http://www.dogsbodynet.com). SONY a développé et mis gratuitement à disposition un outil appelé « Motion Editor » (MEdit) récupérable sur http://openr.aibo.com.

Pour utiliser MEdit, téléchargez-le, puis installez-le. A l'heure actuelle, il n'est disponible que pour le système d'exploitation Windows, mais il fonctionne correctement sous Linux s'il est démarré avec Wine (c'est un émulateur Windows gratuit disponible sur http://www.winehq.com).

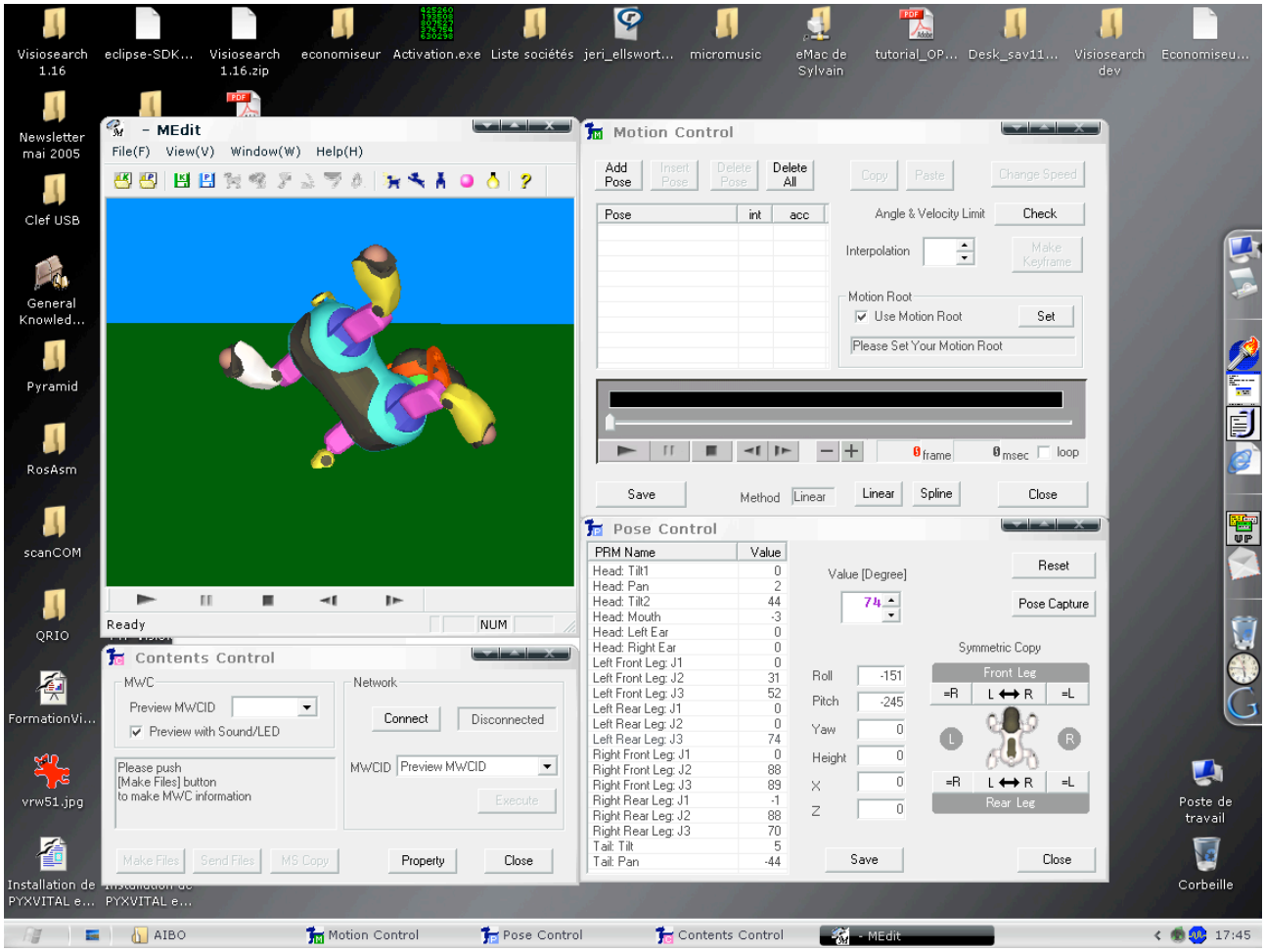

Image 5 – MEdit tournant sous Windows XP

Une fois installé, utilisez MEdit pour créer vos propres mouvements. Reportez-vous au manuel fourni avec le logiciel pour apprendre à bien exploiter ses capacités.

### 4.1.2 – Créer des fichiers audio

Ces fichiers au format WAVE (extension *.wav) ou MIDI (extension *.mid) contiennent les sons et la musique qu'AIBO peut jouer. SONY ne propose aucun outil permettant de créer ce genre de contenu, mais il existe beaucoup d'applications de ce type qui sont téléchargeables gratuitement sur Internet. Certaines contraintes concernant la nature du fichier sont à prendre en compte :

1. Les fichiers WAV peuvent être encodés dans les formats suivants :

   _ 8k 8 bits MONO PCM,

_ 16k 16 bits MONO PCM,
_ 8k 4 bits MONO IMA ADPCM,
_ 16k 4 bits MONO IMA ADPCM.

2. Les fichiers MIDI doivent impérativement être au format SMF.

Si vous ne disposez d'aucun programmme à même de générer ces fichiers, utilisez la fonction intégrée au logiciel « Skitter » évoqué précédemment.

### 4.1.3 – Créer des animations IllumFace

Les fichiers LED (*.led) sont des animations exploitant le système IllumFace intégré dans la tête des AIBO ERS-7. Ils utilisent le même format que les fichiers MIDI, ce qui fait que vous pouvez utiliser votre logiciel de création de musique pour conçevoir des animations IllumFace. N'oubliez pas de les sauvegarder en SMF avec juste une seule piste active. Vous avez également la possibilité d'utiliser Skitter, puisqu'il intègre un éditeur d'animations LED.

## 4.2 – Conversion en fichier ODA

Suivez cette procédure pour convertir vos fichier MTN, MIDI, WAV ou LED en (respectivement) MOTION.ODA, AUDIO.ODA ou LED.ODA :

1. Pour être en mesure d'effectuer la conversion vers le format ODA, vous devez créer un dossier pour chaque type de document : Un répertoire « MOTION », un répertoire « AUDIO » et un répertoire « LED ». Placez les fichiers à convertir dans les bons dossiers, puis lancez le programme RTOOL.EXE disponible sur le site d'OPEN-R (http://openr.aibo.com),

2. Cliquez sur l'onglet ODA et indiquez l'emplacement des dossiers « MOTION », « AUDIO » et « LED » créés à l'étape précédente. Renseignez également le champ « Output Path » qui indique à quel emplacement seront placés les fichiers ODA résultants de la conversion,

3. Choisissez les fichiers ODA que vous voulez générer, puis cliquez sur « Make ODA ».

## 4.3 – Créer le fichier MWC

Le fichier MWC décrit tous les nouveaux mouvements, sons et animations IllumFace que vous avez créé et leur attribue un identifiant (ID). Cet identifiant vous servira à appeler vos créations dans vos programmes en R-Code (PLAY:MWCID:<ID de votre ressource>). Le fichier MWC utilisé par l'ERS-7 s'appelle ERS-7.MWC. Pour créer ce fichier binaire, vous devez tout d'abord créer un fichier texte nommé MWC.CFG et le convertir en utilisant RTOOL. Voici un exemple de ce que contient un fichier MWC :

```
#MWC 1.0
1 3
26000 3
cmagentMOTIONPERFORMER a_stand#stand_so0r_greet 1 1 0x0 0x0 0
cmagentSOUNDPERFORMER soc_d00greetso0r_x1x 1 1 0x0 0x0 0
cmagentFACELIGHT so12_d00greetso0r_l1f 1 1 0x0 0x0 0
```

Voici la marche à suivre pour générer ce fichier :

1. Créer un fichier nommé MWC.CFG. Vous pouvez choisir d'en créer un nouveau ou d'en modifier un qui existe déjà. Si vous optez pour la première option, procédez comme suit :

(a) Chaque nouvelle commande devrait avoir un identifiant unique compris entre 26000 et 27999. C'est ce nombre qui est passé en paramètre à l'instruction *PLAY*,

(b) Le fichier MWC.CFG doit respecter le format suivant :

```
#MWC 1.0
Num1 Num2
Num3 Num4
Text1 Text2 Num5 Num6 Num7 Num8 Num9
```

_ Num1 indique le nombre de commandes (MWCommands) incluses dans le fichier,
_ Num2 indique le nombre de commandes CMAgent incluses dans le fichier,
_ Num3 est l'identifiant (situé entre 26000 et 27999) de la commande décrite à la ligne du dessous,
_ Num4 est le nombre de CMAgent utilisés dans cette même commande,
_ Text1 indique quel opérateur CMAgent (CMAgent Performer) est utilisé. Cela dépend de la partie du corps du robot qui doit être animée :

— **cmagentMOTIONPERFORMER**
-> Pour les mouvements de tout le corps,

— **cmagentMOUTHPERFORMER**
-> Pour les mouvements de la bouche,

— **cmagentHEADPERFORMER**
-> Pour les mouvements de la tête,

— **cmagentLEGPERFORMER**
->Pour les mouvements des jambes,

— **cmagentTAILPERFORMER**
-> Pour les mouvements de la queue,

— **cmagentEARPERFORMER**
-> Pour les mouvements des oreilles,

— **cmagentSOUNDPERFORMER**
-> Pour jouer des sons,

— **cmagentMODELIGHT**

-> Pour gérer les LED qui servent à indiquer dans quel mode se trouve le robot,

— **cmagentFACELIGHT**
-> Pour utiliser les LED de l'IllumFace,

— **cmagentEARLIGHT**
-> Pour gérer les LED du crâne,

— **cmagentBACKLIGHT**
-> Pour utiliser les LED du dos,

— **cmagentLIVELIGHT**
-> Pour gérer la LED indiquant la connexion au réseau Wi-Fi.

_ Text2 est le nom de l'action que le CMAgent doit exécuter,
_ Num5 spécifie combien de fois la commande CMAgent doit être exécutée (généralement, c'est 1),
_ Num6 indique si une synchronisation entre plusieurs actions simultanées est nécessaire (0=non, 1=oui),
_ Num7 est un paramètre destiné au système. Il est toujours à 0x0,
_ Num8 donne des précisions sur la façon dont la commande peut être interrompue. S'il est à 0x01, alors elle peut être stoppée n'importe quand. S'il est à 0x0, cela signifie qu'il faut systématiquement attendre que la commande soit arrivée à son terme. Autrement dit, c'est une action qui ne doit pas être interrompue,
_ Num9 devrait toujours être à 0.

2. Pour commencer la conversion, lancez RTOOL, sélectionnez le fichier MWC.CFG (Input file), indiquez l'emplacement du fichier de sortie (ERS-7.MWC ; Output file) et de BASE-7.MWC (Base file) qui est fourni avec RTOOL dans le dossier MWC.

3. Cliquez sur « Make MWC », le fichier ERS-7.MWC est maintenant créé.

## 4.4 – Copier les fichiers sur le Memory Stick

Une fois tous les fichiers créés, ils doivent être placés sur le Memory Stick de manière à être exploitables dans vos programmes R-Code. Copiez MOTION.ODA, LED.ODA et AUDIO.ODA dans OPEN-R/MW/DATA/P/. Copiez ensuite le fichier ERS-7.MWC dans OPEN-R/MW/CONF/.

Vous êtes maintenant en mesure d'utiliser vos propres ressources dans vos programmes (PLAY:MWCID:<ID de votre ressource>).