Haute Ecole de la Province de Liège : département informatique
==========

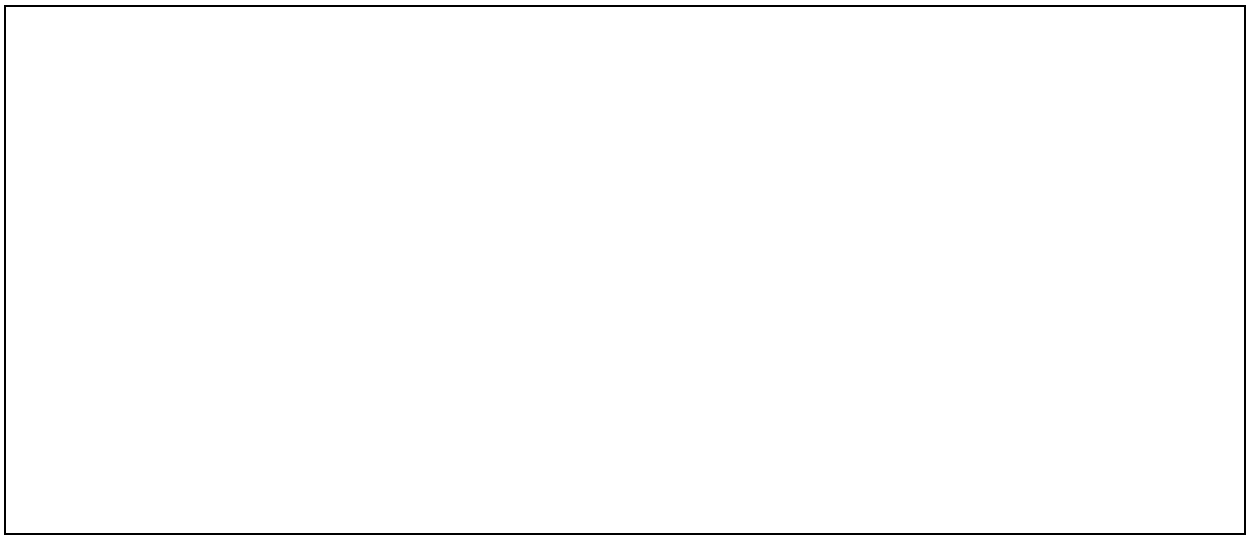
Bachelier en Informatique et Systèmes : finalité Informatique Industrielle

LABORATOIRE D'INFORMATIQUE INDUSTRIELLE
==========

AUTOMATISATION ET ROBOTIQUE laboratoire

2ème année

ROBOT BIOLOID - SCORPION
==========



Année Académique 2014-2015

Titulaire : M. Richard Starzak

VINDERS Romain

TOMBEUR DE HAVAY Hubert

Groupe : 2222

Table des matières

1 - Présentation du dossier

Objectifs

L'objectif principal du projet est de pouvoir commander un robot Bioloid de type Scorpion. Initialement, nous pensions le télécommander via une manette. Toutefois, l'utilisation de cette dernière s'est révélée bien trop compliquée, c'est pourquoi nous avons opté pour un contrôle au clavier, avec raccordement du robot par câble série.

Le programme est développé en C++ à l'aide de Visual Studio. Une librairie permet la communication avec le robot. Ce projet nous aura permis d'apprendre à utiliser plusieurs techniques sous Windows (threads, mutex, priorités, saisie avancée, …).

Le robot effectuera une série d'actions en fonction des touches sur lesquelles on appuie :

- déplacement du robot vers l'avant, rotation vers la gauche ou vers la droite ;
- fermeture des pinces (individuellement) tant que leur touche respective est pressée ;
- déplacement de la tête du scorpion vers le haut ou vers le bas ;
- séquence d'attaque avec la queue du scorpion, avant de reprendre la position initiale ;
- prise de position initiale lors du lancement de l'application.

Organisation

L'utilisation d'un langage orienté objet permet une séparation des différents aspects du programme en différentes classes. Grâce à cela, nous avons pu nous répartir le travail efficacement, ce qui a permis de travailler séparément sur différentes tâches.

Outre l'application principale du robot, plusieurs classes ont donc été créées :

- une classe est dédiée aux entrées-sorties du scorpion. Elle prend en charge l'initialisation du robot, la communication avec celui-ci, ainsi que le déplacement de ses moteurs.
- une classe est dédiée aux séquences : attaque et déplacement. C'est une classe outil, qui n'est pas destinée à être instanciée, mais uniquement utilisée par l'application principale. Elle prend en charge les séquences atomiques (l'attaque) et les séquences fonctionnant par étapes interruptibles (le déplacement). Elle gère également des étapes de transition.

Une classe aurait pu être envisagée pour les actions directes (pinces, tête, …), mais elle n'aurait rien apporté et aurait rendu le code inutilement compliqué.

L'application, elle, s'occupe d'instancier l'objet robot, puis de lancer les différents threads (permettant de séparer plusieurs procédés devant avoir lieu à peu près « simultanément ») :

- saisie régulière de l'activité du clavier (avec priorité supérieure) ;
- exécution des actions directes demandées ;
- exécution des séquences ou étapes de séquences.

2 - Méthode de développement

Outils utilisés

La programmation en C++ du robot Bioloid nécessitait l'utilisation d'une librairie appelée « *Dynamixel SDK* ». Cette librairie permet de communiquer avec le robot. Ainsi, on peut s'en servir pour demander le déplacement d'un moteur (ainsi que toutes ses caractéristiques), ou encore pour contrôler l'état de ce déplacement. Conçue pour Windows, cette librairie a nécessité un développement sous un environnement Windows, ainsi que l'adaptation des techniques utilisées sous UNIX. Vous trouverez plus de détails sur l'utilisation de la librairie au point 4 du rapport (expérimentations).

Afin de gérer efficacement le projet, il fallait un environnement de développement. Nous avons opté pour *Visual Studio*, qui est une référence sous Windows. De plus, ce logiciel a l'avantage d'être présent sur les machines du laboratoire.

D'autres outils ont été utilisés durant les expérimentations. *Robotis RoboPlus Manager* a permis de découvrir le fonctionnement du robot, connecté au moyen d'un câble USB. Ce logiciel permet de mettre le robot en mouvement sans programmation, ainsi que de tester des valeurs et tester l'état de la connexion avec le robot.

Lors de la tentative d'utilisation d'une manette pour télécommander le robot, le logiciel *Eclipse* fut utilisé comme environnement de développement, car il permettait de créer un programme pouvant directement être placé dans la mémoire du robot (ce qui permet une exécution sans avoir besoin de brancher le robot à l'ordinateur).

Enfin, plusieurs câbles sont nécessaires pour l'utilisation du robot. D'abord un câble d'alimentation, afin de fournir du courant au robot. Ensuite un câble le reliant à l'ordinateur de commande. Ce dernier peut être de deux types :

- un simple câble USB/micro-USB permet d'effectuer des tests avec le logiciel *RoboPlus*, et aussi de lancer des animations scriptées ;

- un câble série avec adaptateur USB (côté PC) peut directement être connecté aux servomoteurs. Il permet de programmer le robot en se servant de la librairie *Dynamixel SDK*.

Threads sous Windows

La création et gestion de processus sous Windows est nettement plus compliquée que sous UNIX/Linux et présente pas mal de désavantages. En revanche, l'utilisation des threads est beaucoup plus simple et beaucoup mieux organisée. Pour cette raison, nous avons choisi d'utiliser des threads. L'accès à diverses propriétés telles que la priorité du thread est également très facile sous Windows. De plus, l'utilisation de threads permet de partager des données sans devoir créer de mémoire partagée.

La méthode de création des threads sous Windows est fort similaire à celle utilisée sous UNIX. On appelle la fonction `CreateThread` en lui passant en paramètre toutes les informations nécessaires : nom de la fonction appelée, données transmises, …

Une fois le thread créé, on peut récupérer les données nécessaires. Par mesure de sécurité, lorsqu'aucune donnée ne lui est transmise, on fait appel à `UNREFERENCED_PARAMETER`. Lorsque le thread se termine, il peut renvoyer une valeur de retour (ou zéro s'il ne renvoie rien).

De nombreuses fonctions de manipulation du thread courant sont disponibles. Par exemple, `GetCurrentThread()` permet de récupérer l'identifiant du thread courant, exactement comme `pthread_self()` sous UNIX. On peut facilement modifier la priorité d'un thread, en utilisant la fonction `SetThreadPriority`.

Mutex sous Windows

Notre application partage des données entre plusieurs threads (les demandes faites par l'utilisateur) et l'utilisation des moteurs est également disponible pour plusieurs threads. Pour cette raison, il est nécessaire d'utiliser des sémaphores d'exclusion mutuelle, ou *mutex*.

Les mutex sous Windows sont très similaires aux sémaphores sous UNIX. On crée un mutex à l'aide de `CreateMutex`, et on le supprime en fin d'application avec `CloseHandle`. Les mutex ainsi créés peuvent être nommés ou non nommés (si le troisième paramètre vaut *NULL*). On peut ensuite utiliser des fonctions pour réserver le mutex (`WaitForSingleObject`) et pour le libérer (`ReleaseMutex`), équivalentes au *wait* et au *signall post*.

Astuces de programmation

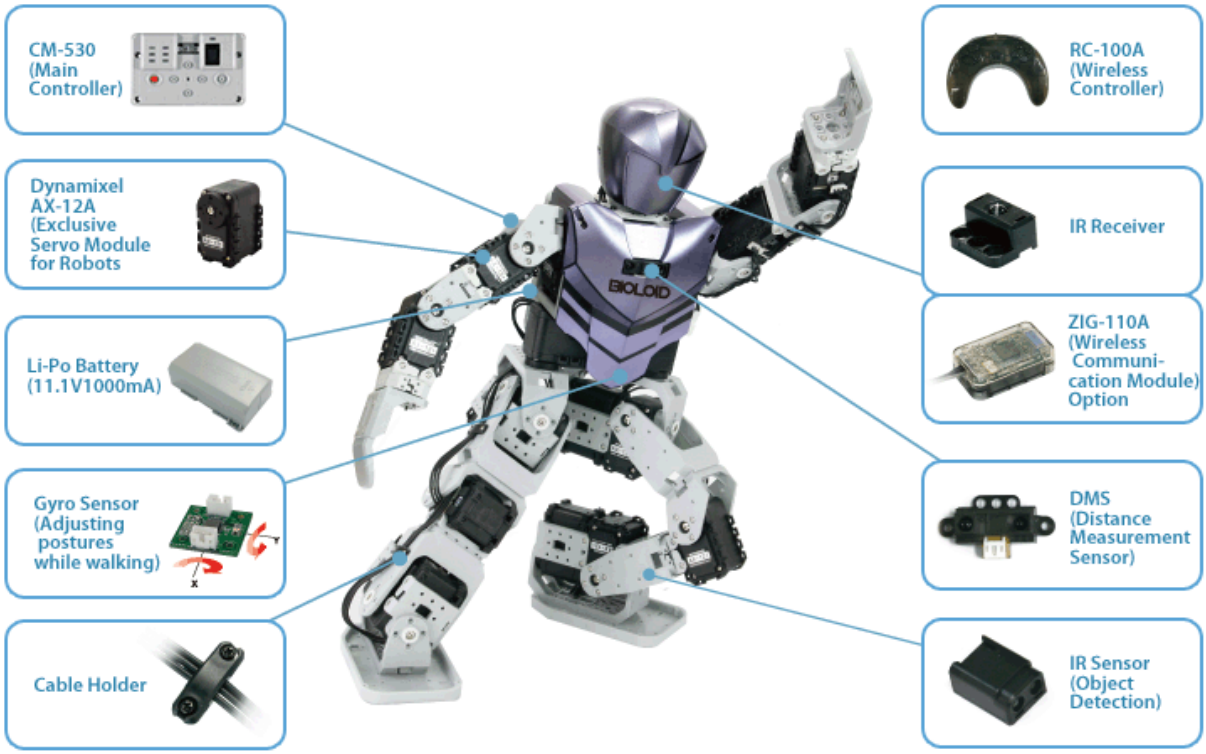
De nombreuses constantes sont utilisées dans le but de fortement simplifier la programmation et de pouvoir rapidement adapter le programme sans toucher au code :

- l'ensemble des identifiants des moteurs du scorpion sont définis dans des constantes portant un nom qui caractérise un moteur. On le différencie ainsi facilement des autres.
- les numéros des séquences et transitions sont définis dans des constantes, ce qui évite de devoir constamment réfléchir à quel numéro désigne quelle séquence.
- la configuration de l'application est définie dans des constantes (paramètres de connexion du robot, table d'adresses, nombre de threads).

3 - Présentation du Bioloid

Le robot Bioloid

Les robots Bioloid (de la firme *Robotis*) sont fournis sous la forme de kits pouvant être assemblés de différentes manières. Ils comportent un microcontrôleur, une batterie, et un ensemble de servomoteurs et de pièces structurelles à assembler.



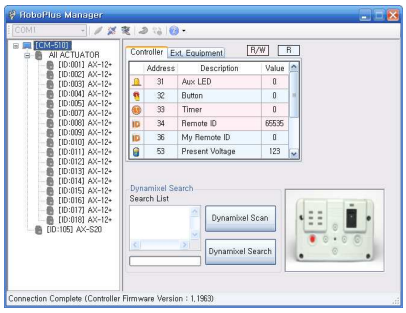
Ces kits peuvent s'assembler de différentes manières, en fonction du but recherché : humanoïdes, chariots, araignées, lézards, chiens, scorpions, dinosaures, …

Ces robots peuvent se commander à distance grâce à une manette utilisant la technologie ZigBee. On peut aussi les connecter à un ordinateur de contrôle au moyen d'un câble.

Méthodes de programmation

Différentes méthodes sont disponibles pour programmer les robots Bioloid. Ci-après sont listées les principales méthodes de programmation.

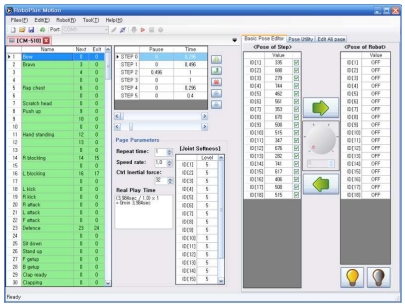
RoboPlus Manager



Ce logiciel permet de facilement tester le robot et les valeurs des moteurs. Il ne s'agit pas réellement de programmation, car l'on est obligé de modifier manuellement les valeurs pour déplacer les moteurs du robot.

RoboPlus Manager est généralement le premier outil utilisé pour découvrir le robot, quelle que soit la méthode choisie ensuite. Le robot doit être connecté en USB.

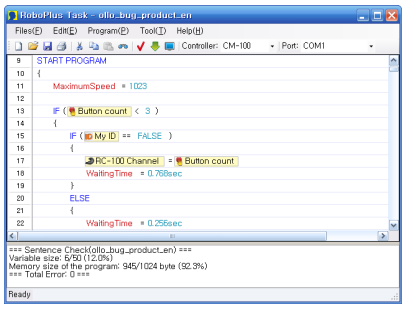
RoboPlus Motion



Ce logiciel permet d'éditer et exécuter des séquences. On ajoute une série d'étapes d'une certaine durée, pour lesquelles on définit la position de chaque servomoteur. On peut aussi avoir un aperçu 3D du robot.

Il s'agit d'une méthode de programmation directe, ne nécessitant aucun langage. Cette méthode est la plus intuitive et serait le choix idéal pour un automaticien.

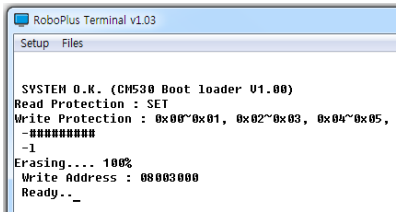
RoboPlus Task



Ce logiciel permet d'utiliser un langage de pseudo-code appelé « *task code* ». Chaque tâche représente un ensemble de déplacements pour effectuer une certaine action. Le robot se déplace selon les tâches codées.

RoboPlus Task permet de créer et gérer facilement ce pseudo-code. Il fournit une série d'outils qui en facilitent l'écriture, ainsi que le choix des moteurs concernés.

RoboPlus Terminal avec programmation en C

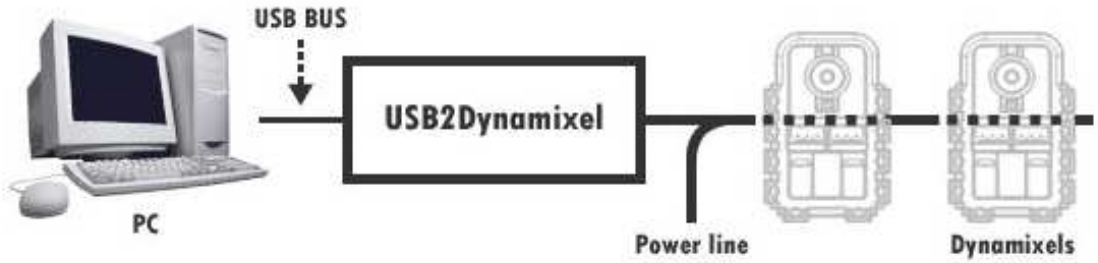


Il est possible d'écrire directement le code exécuté par le microcontrôleur. Le logiciel *RoboPlus Terminal* permet d'effectuer des tests et d'envoyer le code compilé sur le microcontrôleur du robot.

Librairie Dynamixel SDK et programmation sous Windows

Cette méthode est celle que nous avons choisi d'utiliser. Au moyen de la librairie Dynamixel, on communique facilement avec les servomoteurs du robot (*Dynamixel AX-12*), sans devoir se soucier des aspects matériels de très bas niveau. On peut alors choisir le langage de programmation que l'on souhaite utiliser (C, C++, C#, Java, Python, …). Notons qu'un adaptateur USB2Dynamixel sera nécessaire pour connecter le robot au PC.

Schéma de branchement avec USB2Dynamixel



4 - Expérimentations

Découverte et contrôle

La première étape a été de tester le robot en utilisant *RoboPlus Manager*. Nous avons branché le robot à l'ordinateur en utilisant un câble USB/micro-USB, afin de pouvoir s'assurer que chaque moteur fonctionne et vérifier comment cela fonctionne. Chaque moteur a répondu correctement aux positions demandées. Toutefois, nous avons pu constater quelques spécificités du montage.

Nous avons d'abord constaté que les moteurs contrôlant les pattes gauches et droites sont montés symétriquement, ce qui paraît logique. Cela a pour conséquence que les valeurs des positions ne sont pas les mêmes de part et d'autre du robot. En effet, si une valeur déplace une patte droite vers l'avant, la même valeur déplacerait une patte gauche vers l'arrière.

Ensuite, il apparaît que chaque moteur peut prendre une valeur de rotation variant de 0 à 1023. Toutefois, ces limites sont purement théoriques, car en pratique aucun moteur ne peut faire une rotation complète. Il convient de repérer les valeurs maximales et minimales à utiliser pour chaque moteur, afin de ne pas le faire forcer. Si on dépasse la valeur qu'un moteur peut physiquement atteindre, il force légèrement et se met à chauffer. Heureusement, le moteur arrête de forcer après quelques tentatives, ce qui évite d'endommager le matériel. Toutefois, il faut redémarrer l'application pour qu'il accepte à nouveau de bouger.

Pour finir, on constate que l'un des moteurs (la patte droite centrale) utilise un câble trop court. Si on procède à une trop grande rotation, le moteur débranche la patte suivante. Il faut donc garder ce détail à l'esprit lors des tests.

Déplacement des servomoteurs par programmation

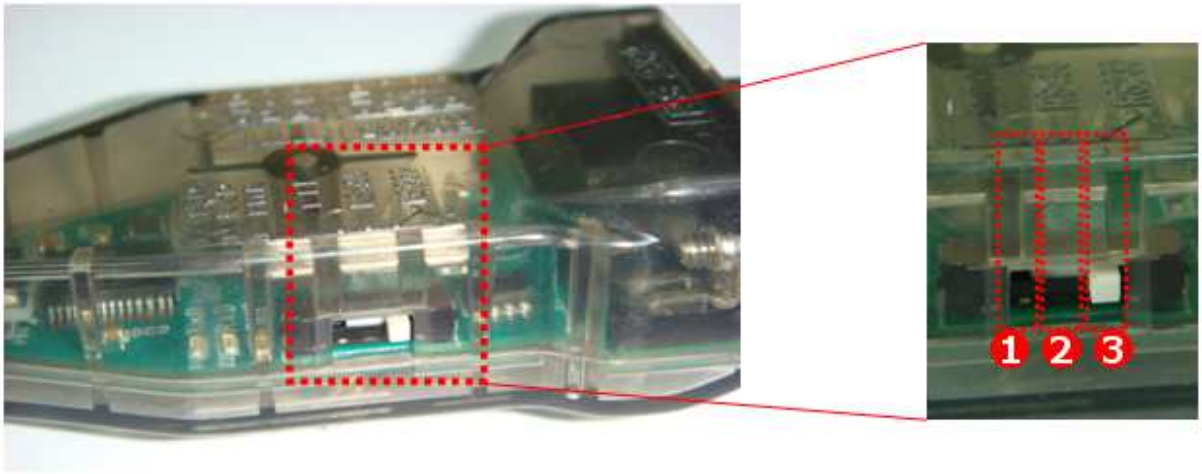
Après avoir testé le fonctionnement du robot, notre but fut de réussir à mettre en mouvement les moteurs du robot en les programmant en C. Pour ce faire, nous avons utilisé la librairie Dynamixel SDK. Celle-ci propose un ensemble de fonctions de lecture et écriture vis-à-vis des servomoteurs du robot.

Nous avons été confrontés à un premier problème qu'il a fallu longtemps pour résoudre : la communication ne semblait pas se faire entre l'ordinateur (donc le programme en C) et le robot. Pourtant, elle fonctionnait avec *RoboPlus Manager*. Le problème venait du câblage. En effet, le câble USB/micro-USB que nous utilisions ne permettait pas l'accès aux servomoteurs, mais uniquement les tests avec le logiciel.

Il fallait donc utiliser un câble série directement connecté aux servomoteurs, ainsi qu'un adaptateur USB2Dynamixel. Monsieur Starzak nous a aidés en nous fournissant le câble en question, ce qui a permis de voir que le code fonctionnait bel et bien.

Fonctionnement de la librairie *Dynamixel SDK*

Avant de pouvoir utiliser la librairie, il faut d'abord un câblage approprié pour relier le robot à l'ordinateur. Il est également nécessaire que l'adaptateur USB2Dynamixel soit configuré en mode 3 (RS232) grâce à un mini-switch, comme le montre l'image qui suit.



L'utilisation de la librairie s'effectue en trois étapes : initialisation, utilisation, fermeture.

Pour effectuer l'initialisation, il suffit de faire appel à une fonction de la librairie prévue pour ça (`dxl_initialize`). Elle se chargera de configurer la librairie pour contrôler correctement les servomoteurs. On doit lui fournir le *baud-rate*, ainsi que le numéro du port USB utilisé pour communiquer avec le robot.

Pour utiliser les servomoteurs, des fonctions de lecture/écriture sont disponibles. Pour contrôler un servomoteur, il faut fournir son identifiant (numéro), préciser le type de contrôle (par rapport au tableau de contrôle) et enfin la valeur souhaitée. Par exemple, on peut modifier le servo-moteur 1 en plaçant sa position à une valeur entre 0 et 1023. On pourrait aussi définir, pour le servomoteur 1, sa vitesse selon une valeur précise.

On peut également effectuer une lecture, afin de connaître l'état actuel d'un moteur. Comme dans le cas de l'écriture, on doit préciser l'identifiant du moteur, ainsi que le type de valeur qu'on veut lire (la position, la vitesse, …).

Enfin, lorsque l'on ne souhaite plus utiliser le robot, il convient de terminer correctement la communication entre lui et la librairie. Pour ce faire, il existe une fonction (`dxl_terminate`).

Pour mieux comprendre la librairie Dynamixel SDK, nous avons principalement consulté deux pages web du site officiel de Robotis :

http://support.robotis.com/en/techsupport_eng.htm#software/dynamixel_sdk/usb2dynamixel/window_communication_2/visual_c++_2.htm

http://support.robotis.com/en/product/dynamixel/rx_series/rx-28.htm

Utilisation de la manette

La dernière étape avait pour buts de pouvoir contrôler le robot à distance avec la manette et de pouvoir le rendre indépendant de l'ordinateur pour ne plus devoir y être branché. Après maintes recherches pour faire de la programmation embarquée, il s'est avéré que cela nécessitait l'utilisation d'un autre environnement de développement que *Visual Studio*. Nous avons opté pour *Eclipse* car il permet la programmation en C et C++.

Le premier test consistait à réussir à injecter un programme dans le robot pour qu'il l'exécute. Notre programme se contentait d'ouvrir et fermer les pinces du robot, le but étant juste de réussir à le contrôler de l'intérieur. Le test fut concluant.

Le second test devait faire interagir la manette sans-fil avec le robot. Pour ce faire, une autre librairie était nécessaire : ZigBee SDK. Son fonctionnement est fort similaire à celui de Dynamixel SDK et nécessite les mêmes étapes. Le programme de ce second test allumait une LED du robot lorsqu'une touche de la manette était pressée. Le résultat fut à nouveau concluant. D'autres tests ont permis d'utiliser plusieurs LEDs selon la touche pressée.

Le dernier test avait pour but d'utiliser les deux librairies simultanément, afin de contrôler les moteurs selon les indications de la manette. Nous ne sommes pas parvenus à faire cohabiter les deux librairies, donc le résultat ne fut pas concluant cette fois.

Plusieurs manières de procéder ont été testées pour ce dernier test :

- attendre qu'une touche soit pressée. La touche A allumerait une LED, la touche B fermerait une pince. Tant que la touche B n'était pas pressée, tout semblait fonctionner. Hélas, dès que la touche B était pressée et que les moteurs devaient se mettre en marche, le programme s'arrêtait. La librairie Dynamixel SDK ne fonctionnait pas.
- faire bouger les pinces en continu en attendant qu'une touche soit pressée. Une fois une touche pressée, les pinces devaient s'arrêter et la tête devait bouger. Tout fonctionnait jusqu'au moment où une touche était pressée. La librairie ZigBee SDK ne fonctionnait pas.

Après de longues recherches dans le code d'initialisation des librairies, il s'est avéré que les deux librairies utilisaient le canal USART2. Nous avons tenté en vain d'utiliser le canal USART3 avec la librairie Dynamixel SDK, mais il semble que ce canal n'est que théorique et n'est pas présent pour ce robot. Vu qu'il n'existait pas, il était donc impossible de l'utiliser.

D'autres tentatives d'ajustement des autres paramètres n'ont rien donné non plus. Nous avons donc abandonné l'idée de la manette. L'utilisation du clavier de l'ordinateur pour le contrôle était donc la seule alternative. Le câblage vers l'ordinateur étant à présent nécessaire pour le contrôle, il ne fallait donc plus injecter le code sur le microcontrôleur.

5 - Schéma de l'application

Diagramme fonctionnel

Après l'initialisation, le programme se sépare en trois threads. Le thread principal devient un thread de lecture de l'activité du clavier. Il permet de mettre régulièrement à jour les données partagées qui indiquent les demandes d'actions/séquences. Il bénéficie d'une priorité supérieure aux autres threads. C'est lui qui gère la demande d'arrêt du programme.

Le second thread lit les données partagées, afin de déterminer les demandes de séquences. Il exécute ensuite la plus prioritaire des séquences demandées (la priorité étant donnée à la séquence d'attaque, puis à la rotation et en dernier lieu au déplacement vers l'avant). La séquence d'attaque est exécutée dans son ensemble et on ne peut l'interrompre. La séquence de marche/rotation est cyclique et séparée en étapes individuelles. On exécute une étape à la fois et on mémorise la prochaine étape dans une variable statique.

Le dernier thread lit les données partagées, afin de déterminer les demandes d'actions simples. Ces actions peuvent être un déplacement de la tête ou une ouverture/fermeture de pince. L'ensemble des actions simples demandées est exécuté à chaque tour de boucle.

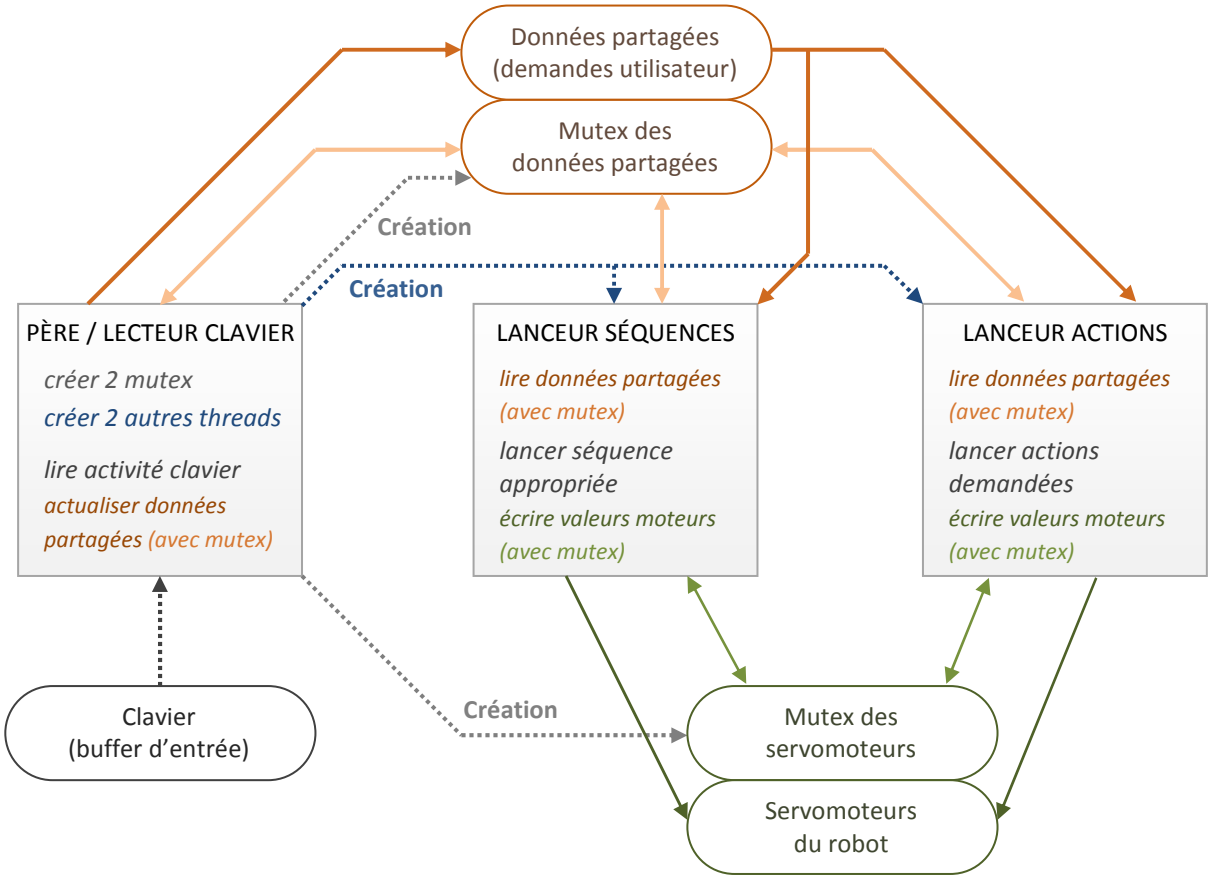
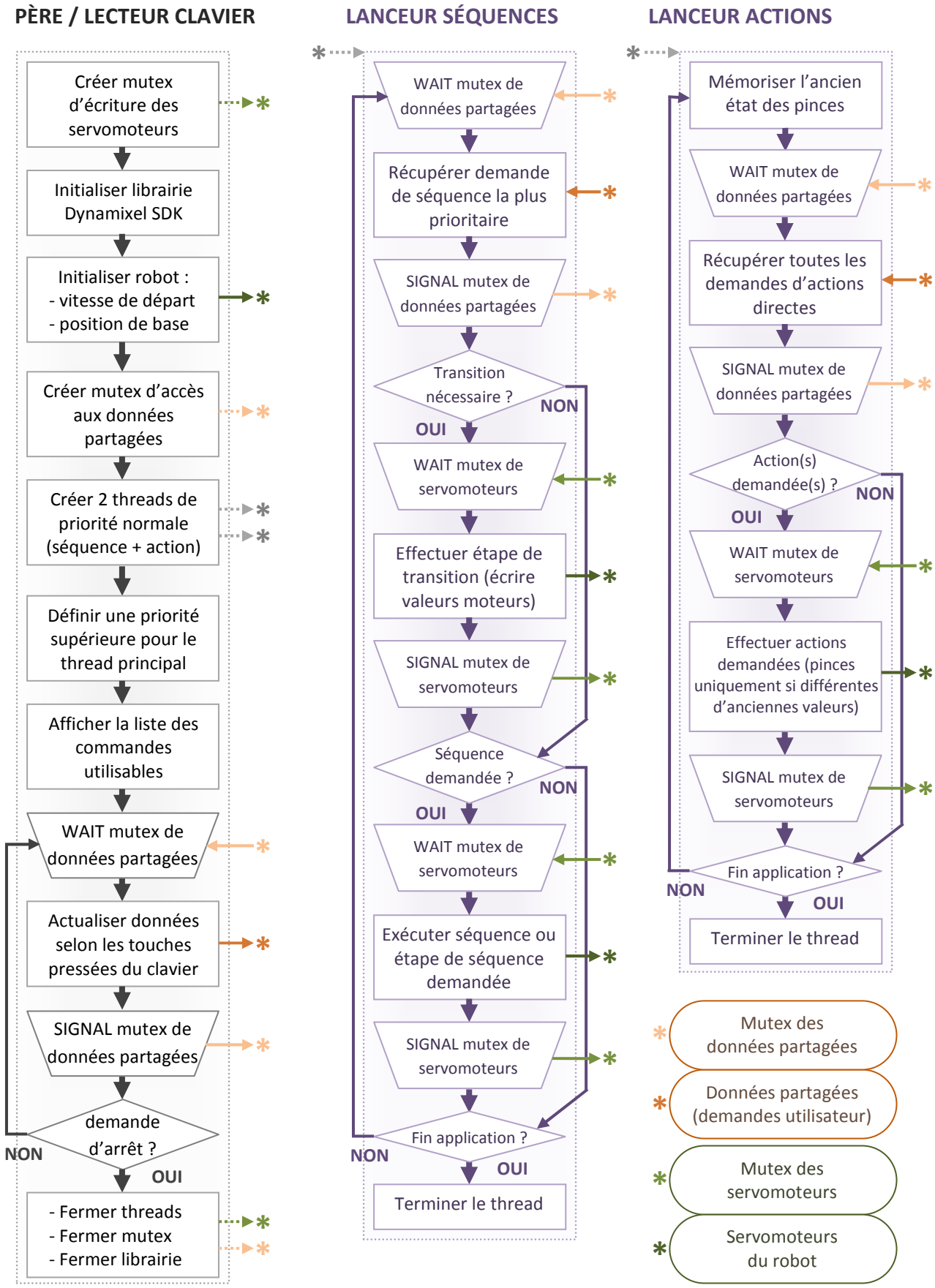


Schéma des processus



6 - Code de l'application

bioloid_app.cpp

```
/*************************************************************
VINDERS Romain & TOMBEUR DE HAVAY Hubert - 2222
BIOLOID - SCORPION - Application de contrôle
Date : 02/04/2015
*************************************************************/

#include <iostream>
#include <stdlib.h>
#include <climits>
#include <windows.h>

using namespace std;
#include "bioloid_sequence.h"
#include "bioloid_scorpion.h"

#define NB_THREADS 2  //pour déclaration et fermeture



/* prototypes - application de contrôle */
void readKeyboard();
DWORD WINAPI runSequence(LPVOID);
DWORD WINAPI runAction(LPVOID);
void displayCommands();
int getSequence(int, int*);
void getAction(short*, short*);
void setAction(int, int);
void closeApplication(int);


/* structure données partagées (input) */
struct
{
    short front; //déplacement
    short left;
    short right;
    short attack; //attaque
    short head_up; //tête
    short head_down;
    short pincer_left; //pinces
    short pincer_right;
} pressedKeys;


/* variables de contrôle */
bool appContinue = true;
HANDLE hThreadArray[NB_THREADS];
Scorpion *robot;       //outil d'entrée-sortie du robot
HANDLE hMutexReadKeys; //mutex données partagées


/*-------------------------------------------------------------------------*/
```

```
int main()
{
    /* initialisation robot */
    robot = new Scorpion(140);

    /* création mutex données partagées */
    hMutexReadKeys = CreateMutex(NULL, FALSE, NULL); //mutex données partagées (input)
    if (hMutexReadKeys == NULL)
    {
        cout << "MAIN: Erreur creation mutex lecture" << endl;
        exit(1);
    }

    /* création des threads */
    hThreadArray[0] = CreateThread(NULL,0, runSequence, NULL,0,NULL);//séquences
    hThreadArray[1] = CreateThread(NULL,0, runAction, NULL,0,NULL);  //actions simples
    if (hThreadArray[0] == NULL || hThreadArray[1] == NULL)
    {
        cout << "MAIN: Erreur creation threads" << endl;
        appContinue = false;
        closeApplication(1);
    }

    /* thread principal - lecture clavier */
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_ABOVE_NORMAL); //priorité+1
    readKeyboard();

    /* attente fin threads et fermeture */
    closeApplication(0);
    return 0;
}

/*-----------------------------------------------------------------------------
  THREAD LECTURE DU CLAVIER (INPUT)
 -----------------------------------------------------------------------------*/
void readKeyboard()
{
    displayCommands(); //afficher mode d'emploi

    while (appContinue)
    {
        Sleep(30);
        if (GetAsyncKeyState(VK_ESCAPE) ) //demande de fermeture
            appContinue = false;
        else
        {
            if (WaitForSingleObject(hMutexReadKeys,INFINITE) == WAIT_OBJECT_0) //mutex input
            {
                /* lire et actualiser input clavier */
                pressedKeys.front = GetKeyState(VK_UP) & 0x8000;
                pressedKeys.left = GetKeyState(VK_LEFT) & 0x8000;
                pressedKeys.right = GetKeyState(VK_RIGHT) & 0x8000;
                pressedKeys.attack = GetKeyState(VK_SPACE) & 0x8000;
                pressedKeys.head_up = GetKeyState(0x5A) & 0x8000; //Z
                pressedKeys.head_down = GetKeyState(0x53) & 0x8000; //S
                pressedKeys.pincer_left = GetKeyState(0x51) & 0x8000; //Q
                pressedKeys.pincer_right = GetKeyState(0x44) & 0x8000; //D

                if (!ReleaseMutex(hMutexReadKeys) ) //libération mutex input
                {
                    cout << "READKEYBOARD: Erreur liberation mutex(readkeys)" << endl;
                    appContinue = false;
                    closeApplication(1);
                }
            }
        }
    }
}
```

```
/*-------------------------------------------------------------------------
  AFFICHAGE MODE D'EMPLOI
 -------------------------------------------------------------------------*/
void displayCommands()
{
    cout << "------------------------------" << endl;
    cout << "DOSSIER BIOLOID - Scorpion" << endl;
    cout << "par VINDERS R. et TOMBEURDEHAVAY H., 2222" << endl;
    cout << "------------------------------" << endl;
    cout << "\t<Haut/Gauche/Droite>   Deplacer" << endl;
    cout << "\t<Z/S>    Incliner tete" << endl;
    cout << "\t<Q>      Fermer pince gauche" << endl;
    cout << "\t<D>      Fermer pince droitee" << endl;
    cout << "\t<ESPACE> Attaquer" << endl << endl;
    cout << "Appuyez sur ESC pour arreter le programme." << endl;
}



/*-------------------------------------------------------------------------
  THREAD LANCEMENT DE SEQUENCES
 -------------------------------------------------------------------------*/
DWORD WINAPI runSequence(LPVOID lpParam)
{
    UNREFERENCED_PARAMETER(lpParam); //aucun paramètre de thread
    int seqType = 0;
    int transition = 0;

    while (appContinue)
    {
        Sleep(30);

        /* récupérer demande de séquence (prioritaire) */
        seqType = getSequence(seqType, &transition);

        /* transition vers séquence */
        if (transition)
        {
            Sequence::doTransition(robot, transition);
            transition = 0;
        }
        /* lancer séquence demandée */
        if (seqType == SEQ_ATTACK) //séquence d'attaque (complète)
        {
            Sequence::attack(robot);
            seqType = 0;
        }
        else
        {
            if (seqType >= SEQ_WALK_FWD) //séquence de marche (une étape)
                Sequence::walk(robot, seqType);
        }
    }
    return 0;
}
```

```
/*-------------------------------------------------------------------------
  THREAD LANCEMENT D'ACTIONS SIMPLES
 -------------------------------------------------------------------------*/
DWORD WINAPI runAction(LPVOID lpParam)
{
    UNREFERENCED_PARAMETER(lpParam); //aucun paramètre de thread
    short actionHead = 0;
    short actionPincer[2] = {0,0};
    short lastActionPincer[2] = {0,0};

    while (appContinue)
    {
        Sleep(30);

        /* récupérer demande(s) d'action(s) */
        lastActionPincer[0] = actionPincer[0]; //mémoriser précédentes
        lastActionPincer[1] = actionPincer[1];
        getAction(&actionHead, actionPincer); //récupérer actions

        /* déplacer tête selon demande */
        if (actionHead)
        {
            if (actionHead > 0)
                setAction(HEAD_ROTATE, 500);
            else
                setAction(HEAD_ROTATE, 700);
            actionHead = 0; //fin action tête
        }

        /* déplacer pince gauche selon demande */
        if (actionPincer[0] != lastActionPincer[0])
        {
            if (actionPincer[0])
                setAction(PINCER_L, 400);
            else
                setAction(PINCER_L, 600);
        }

        /* déplacer pince droite selon demande */
        if (actionPincer[1] != lastActionPincer[1])
        {
            if (actionPincer[1])
                setAction(PINCER_R, 600);
            else
                setAction(PINCER_R, 400);
        }
    }
    return 0;
}
```

```
/*-------------------------------------------------------------------------
  DETERMINER DEMANDE DE SEQUENCE (SELON INPUT)
 -------------------------------------------------------------------------*/
int getSequence(int lastSeqType, int *pTransition)
{
    int seqType = 0;
    int turn = 0;

    if (WaitForSingleObject(hMutexReadKeys, INFINITE) == WAIT_OBJECT_0) //mutex input
    {
        /* vérifier demande d'attaque */
        if (pressedKeys.attack) //prioritaire
        {
            seqType = SEQ_ATTACK;
        }
        else
        {
            /* vérifier demande de déplacement */
            if (pressedKeys.front != 0) //avancer
            {
                seqType = SEQ_WALK_FWD;
            }
            if ( (turn = pressedKeys.left + pressedKeys.right) != 0) //tourner
            {
                /* s'assurer qu'une seule des deux touches soit pressée */
                if (pressedKeys.left == turn)
                    seqType = SEQ_WALK_LEFT;
                else if (pressedKeys.right == turn)
                    seqType = SEQ_WALK_RIGHT;
            }
        }

        if (!ReleaseMutex(hMutexReadKeys) ) //libération mutex input
        {
            cout << "GETSEQUENCE: Erreur liberation mutex(readkeys)" << endl;
            appContinue = false;
        }

        /* vérifier nécessité de transition */
        if (lastSeqType >= SEQ_WALK_FWD) //fin de marche
        {
            if (seqType == 0 || seqType == SEQ_ATTACK)
                *pTransition = TR_WALK_END;
        }
        else if (!lastSeqType && seqType >= SEQ_WALK_FWD) //début de marche
        {
            *pTransition = TR_WALK_START;
        }
    }
    return seqType;
}
```

```
/*-------------------------------------------------------------------------
  DETERMINER DEMANDE D'ACTION (SELON INPUT)
 -------------------------------------------------------------------------*/
void getAction(short *pHead, short *pPincer)
{
    if (WaitForSingleObject(hMutexReadKeys,INFINITE) == WAIT_OBJECT_0) //mutex input
    {
        /* vérifier demande déplacement tête */
        if (pressedKeys.head_up)
        {
            if (!pressedKeys.head_down)
                *pHead = 1;
        }
        else
        {
            if (pressedKeys.head_down)
                *pHead = -1;
        }

        /* vérifier demande déplacement pinces */
        if (pressedKeys.pincer_left) //pince gauche
            pPincer[0] = 1;
        else
            pPincer[0] = 0;
        if (pressedKeys.pincer_right) //pince droite
            pPincer[1] = 1;
        else
            pPincer[1] = 0;

        if (!ReleaseMutex(hMutexReadKeys) ) //libération mutex input
        {
            cout << "GETACTION: Erreur liberation mutex(readkeys)" << endl;
            appContinue = false;
        }
    }
}


/*-------------------------------------------------------------------------
  EFFECTUER ACTION (AVEC MUTEX SCORPION)
 -------------------------------------------------------------------------*/
void setAction(int motor, int val)
{
    if(WaitForSingleObject(robot->hMutexOutput,INFINITE) == WAIT_OBJECT_0)//mutex IO scorp.
    {
        robot->moveMotor(motor, val, 0); //effectuer action

        if (!ReleaseMutex(robot->hMutexOutput) ) //libération mutex IO scorpion
        {
            cout << "SETACTION: Erreur liberation mutex(output)" << endl;
            appContinue = false;
        }
    }
}
```

```
/*-------------------------------------------------------------------------
  FERMETURE APPLICATION
 -------------------------------------------------------------------------*/
void closeApplication(int retVal)
{
    /* fermeture threads */
    for (int i = 0; i < NB_THREADS; i++)
    {
        if (hThreadArray[i] != NULL)
        {
            WaitForSingleObject(hThreadArray[i], 2000);
            CloseHandle(hThreadArray[i]);
        }
    }

    /* fermeture mutex et robot */
    CloseHandle(hMutexReadKeys);
    delete robot;
    robot = NULL;

    system("pause");
    exit(retVal);
}
```

bioloid_sequence.h

```
/*************************************************************
BIOLOID - SCORPION - Classe outil séquence (marche / attaque)
*************************************************************/

#ifndef BIOLOID_SEQUENCE_H
#define BIOLOID_SEQUENCE_H

#include "bioloid_scorpion.h"

/* numéros de séquences */
#define TR_WALK_START 1
#define TR_WALK_END 2
#define SEQ_ATTACK 1
#define SEQ_WALK_FWD 2
#define SEQ_WALK_LEFT 4
#define SEQ_WALK_RIGHT 8


/* - exécution de séquences - classe outil (non instanciable) - */
class Sequence
{
  private:
    Sequence() {} //bloquer instanciation

  public:
    static int _seqStep;               //étape courante
    static short _steps[4][6];         //valeurs de fin d'étapes
    static short _stepMoves[12][2][2]; //valeurs de déplacements d'étapes

    /* méthodes de séquences */
    static void attack(Scorpion*);
    static void walk(Scorpion*, int);
    static void doTransition(Scorpion*, int);
};

#endif
```

bioloid_sequence.cpp

```
/*************************************************************
BIOLOID - SCORPION - Classe outil séquence (marche / attaque)
*************************************************************/

#include <iostream>
#include <climits>
#include <string.h>
#include <windows.h>

using namespace std;
#include "bioloid_sequence.h"
#include "bioloid_scorpion.h"


/* valeurs particulières */
#define LEG_LALL_ROTATE -1
#define LEG_RALL_ROTATE -2

/* variable de contrôle */
extern bool appContinue;



/* initialisation variables statiques */
int Sequence::_seqStep = 0;              //étape courante
short Sequence::_steps[4][6] = {         //valeurs de fin d'étapes ([étape][patte])
        {400, 620, 620, 620, 620, 620},
        {400, 400, 620, 400, 400, 400},
        {400, 400, 400, 620, 400, 400},
        {620, 620, 620, 620, 620, 400}
    };
short Sequence::_stepMoves[12][2][2] = { //valeurs de déplacements d'étapes
        {{LEG_L1_BEND,420},    {LEG_R3_BEND,600}},  //[partie][côté]{moteur, valeur}
        {{LEG_L1_ROTATE,400}, {LEG_R3_ROTATE,620}},
        {{LEG_L1_BEND,520},    {LEG_R3_BEND,500}},

        {{LEG_L2_BEND,420},    {0,0}},
        {{LEG_L2_ROTATE,400}, {LEG_RALL_ROTATE,400}},
        {{LEG_L2_BEND,520},    {0,0}},

        {{LEG_L3_BEND,420},    {LEG_R1_BEND,600}},
        {{LEG_L3_ROTATE,400}, {LEG_R1_ROTATE,620}},
        {{LEG_L3_BEND,520},    {LEG_R1_BEND,500}},

        {{0,0},                {LEG_R2_BEND,600}},
        {{LEG_LALL_ROTATE,620}, {LEG_R2_ROTATE,620}},
        {{0,0},                {LEG_R2_BEND,500}}
    };
```

```
/*-------------------------------------------------------------------------
  SEQUENCE COMPLETE - ATTAQUE
 -------------------------------------------------------------------------*/
void Sequence::attack(Scorpion *robot)
{
    if(WaitForSingleObject(robot->hMutexOutput,INFINITE) == WAIT_OBJECT_0)//mutex IO scorp.
    {
        /* 1 - incliner */
        robot->setBasePosition(false);
        robot->moveMotor(TAIL_BASE, 700, 0);
        robot->moveMotor(LEG_R1_BEND, 640, 0);
        robot->moveMotor(LEG_R2_BEND, 570, 0);
        robot->moveMotor(LEG_L1_BEND, 380, 0);
        robot->moveMotor(LEG_L2_BEND, 450, 150);

        /* 2 - queue en retrait */
        robot->moveMotor(TAIL_MID, 740, 0);
        robot->moveMotor(TAIL_END, 600, 150);

        /* 3 - attaque */
        robot->setSpeed(160);
        robot->moveMotor(TAIL_BASE, 800, 0);
        robot->moveMotor(TAIL_MID, 780, 0);
        robot->setSpeed(140);
        robot->moveMotor(TAIL_END, 700, 350);

        /* 4 - retour état initial */
        robot->setBasePosition(true);

        if (!ReleaseMutex(robot->hMutexOutput) ) //libération mutex IO scorpion
        {
            cout << "SEQUENCE(attack): Erreur liberation mutex(output)" << endl;
            appContinue = false;
        }
    }
}

/*-------------------------------------------------------------------------
  ETAPE DE SEQUENCE - MARCHE
 -------------------------------------------------------------------------*/
void Sequence::walk(Scorpion *robot, int type)
{
    int lastSeqStep;
    int waitEnd;
    int j;

    if(WaitForSingleObject(robot->hMutexOutput,INFINITE) == WAIT_OBJECT_0)//mutex IO scorp.
    {
        lastSeqStep = _seqStep - 1;
        if (lastSeqStep < 0)
            lastSeqStep = 3;

        /* 1 - configurer position pré-requise */
        if (type != SEQ_WALK_LEFT) //côté gauche (si avant/droite)
        {
            robot->moveMotor(LEG_L1_ROTATE, _steps[lastSeqStep][0], 0);
            robot->moveMotor(LEG_L2_ROTATE, _steps[lastSeqStep][1], 0);
            robot->moveMotor(LEG_L3_ROTATE, _steps[lastSeqStep][2], 0);
        }

        if (type != SEQ_WALK_RIGHT) //côté droit (si avant/gauche)
        {
            robot->moveMotor(LEG_R1_ROTATE, _steps[lastSeqStep][3], 0);
            robot->moveMotor(LEG_R2_ROTATE, _steps[lastSeqStep][4], 0);
            robot->moveMotor(LEG_R3_ROTATE, _steps[lastSeqStep][5], 0);
        }
```

```
        /* 2 - effectuer étape actuelle déplacement (en 3 parties) */
        for (j = 0; j < 3; j++)
        {
            //côté gauche : si vers avant/droite (bloqué si on tourne à gauche)
            if (type == SEQ_WALK_RIGHT || type == SEQ_WALK_FWD)
            {
                if (_stepMoves[_seqStep*3 + j][0][0]) //partie d'étape non vide
                {
                    if (_stepMoves[_seqStep*3 + j][0][0] == LEG_LALL_ROTATE)//rotat. totale
                    {
                        robot->moveMotor(LEG_L3_ROTATE, _stepMoves[_seqStep*3+j][0][1], 0);
                        robot->moveMotor(LEG_L2_ROTATE, _stepMoves[_seqStep*3+j][0][1], 0);
                        robot->moveMotor(LEG_L1_ROTATE, _stepMoves[_seqStep*3+j][0][1], 300);
                    }
                    else //mouvement d'une seule patte
                    {
                        if (_seqStep==0 || (_seqStep==1 && j!=1) || type!=SEQ_WALK_FWD)
                            waitEnd = 150; //partie bloquante
                        else
                            waitEnd = 0; //non bloquante (l'autre côté bloquera)
                        robot->moveMotor(_stepMoves[_seqStep*3 + j][0][0],
                                         _stepMoves[_seqStep*3 + j][0][1], waitEnd);
                    }
                }
            }

            //côté droit : si vers avant/gauche (bloqué si on tourne à droite)
            if (type == SEQ_WALK_LEFT || type == SEQ_WALK_FWD)
            {
                if (_stepMoves[_seqStep*3 + j][1][0]) //partie d'étape non vide
                {
                    if (_stepMoves[_seqStep*3 + j][1][0] == LEG_RALL_ROTATE)//rotat. totale
                    {
                        robot->moveMotor(LEG_R3_ROTATE, _stepMoves[_seqStep*3+j][1][1], 0);
                        robot->moveMotor(LEG_R2_ROTATE, _stepMoves[_seqStep*3+j][1][1], 0);
                        robot->moveMotor(LEG_R1_ROTATE, _stepMoves[_seqStep*3+j][1][1], 300);
                    }
                    else //mouvement d'une seule patte
                    {
                        if (_seqStep==2 || (_seqStep==3 && j!=1) || type!=SEQ_WALK_FWD)
                            waitEnd = 150; //partie bloquante
                        else
                            waitEnd = 0; //non bloquante (l'autre côté bloquera)
                        robot->moveMotor(_stepMoves[_seqStep*3 + j][1][0],
                                         _stepMoves[_seqStep*3 + j][1][1], waitEnd);
                    }
                }
            }
        }

        if (!ReleaseMutex(robot->hMutexOutput) ) //libération mutex IO scorpion
        {
            cout << "SEQUENCE(walk): Erreur liberation mutex(output)" << endl;
            appContinue = false;
            return;
        }

        /* 3 - incrémenter numéro d'étape */
        _seqStep++;
        if (_seqStep > 3)
            _seqStep = 0;
    }
}
```

```
/*-------------------------------------------------------------------------
  TRANSITION AVANT/APRES SEQUENCE
 -------------------------------------------------------------------------*/
void Sequence::doTransition(Scorpion *robot, int type)
{
    if(WaitForSingleObject(robot->hMutexOutput,INFINITE) == WAIT_OBJECT_0)//mutex IO scorp.
    {
        /* choix transition */
        switch (type)
        {
            case TR_WALK_START: //début marche
                _seqStep = 0;
                robot->moveMotor(LEG_L1_BEND, 420, 0);
                robot->moveMotor(LEG_R2_BEND, 600, 100);
                robot->moveMotor(LEG_L1_ROTATE, 510, 0);
                robot->moveMotor(LEG_R1_ROTATE, 620, 100);
                robot->moveMotor(LEG_R2_ROTATE, 620, 150);
                robot->moveMotor(LEG_L1_BEND, 520, 0);
                robot->moveMotor(LEG_R2_BEND, 500, 150);
                break;

            case TR_WALK_END: //fin marche
                robot->setBasePosition(true);
                break;
        }

        if (!ReleaseMutex(robot->hMutexOutput) ) //libération mutex IO scorpion
        {
            cout << "SEQUENCE(transition): Erreur liberation mutex(output)" << endl;
            appContinue = false;
            return;
        }
    }
}
```

bioloid_scorpion.h

```
/************************************************************
BIOLOID - SCORPION - Classe entrée-sortie scorpion
************************************************************/
#ifndef BIOLOID_SCORPION_H
#define BIOLOID_SCORPION_H

#pragma once
#include "dynamixel.h"
#include <windows.h>
#pragma comment(lib, "dynamixel.lib")

/* numéros des moteurs */
#define PINCER_L            1
#define PINCER_R            2
#define TAIL_BASE           16
#define TAIL_MID            17
#define TAIL_END            18
#define LEG_L1_ROTATE       3
#define LEG_L2_ROTATE       7
#define LEG_L3_ROTATE       11
#define LEG_R1_ROTATE       4
#define LEG_R2_ROTATE       8
#define LEG_R3_ROTATE       12
#define LEG_L1_BEND         5
#define LEG_L2_BEND         9
#define LEG_L3_BEND         13
#define LEG_R1_BEND         6
#define LEG_R2_BEND         10
#define LEG_R3_BEND         14
#define HEAD_ROTATE         15

/* table adresse de contrôle */
#define P_GOAL_POSITION_L       30
#define P_GOAL_POSITION_H       31
#define P_PRESENT_POSITION_L    36
#define P_PRESENT_POSITION_H    37
#define P_PRESENT_SPEED_L       38
#define P_PRESET_SPEED_H        39
#define P_MOVING                46

/* configuration de la connexion */
#define DEFAULT_PORTNUM    5 //COM5
#define DEFAULT_BAUDNUM    1 //1Mbps


/* - classe IO moteurs du scorpion - */
class Scorpion
{
  private:
    int _speed; //vitesse des moteurs

  public:
    HANDLE hMutexOutput; //mutex IO scorpion

    /* constructeur / destructeur */
    Scorpion(int);
    ~Scorpion();

    /* méthodes du scorpion */
    void setSpeed(int);
    int  getSpeed();
    void setBasePosition(bool);
    void moveMotor(int, int, int);
};

#endif
```

bioloid_scorpion.cpp

```
/*************************************************************
BIOLOID - SCORPION - Classe entrée-sortie scorpion
*************************************************************/

#include <iostream>
#include <stdlib.h>
using namespace std;
#include "bioloid_scorpion.h"


/*-------------------------------------------------------------------------
  CONSTRUCTEUR - INITIALISATION ROBOT + MUTEX
 -------------------------------------------------------------------------*/
Scorpion::Scorpion(int initSpeed)
{
    /* créer mutex d'accès aux moteurs du scorpion */
    hMutexOutput = CreateMutex(NULL, FALSE, NULL);
    if (hMutexOutput == NULL)
    {
        cout << "SCORPION: Erreur creation mutex pour ecriture" << endl;
        system("pause");
        exit(1);
    }

    /* configuration connexion robot */
    if (dxl_initialize(DEFAULT_PORTNUM, DEFAULT_BAUDNUM) == 0)
    {
        cout << "SCORPION: Erreur ouverture USB2Dynamixel!" << endl;
        CloseHandle(hMutexOutput);
        system("pause");
        exit(1);
    }

    _speed = initSpeed; //vitesse initiale

    /* mise en position de départ */
    if (WaitForSingleObject(hMutexOutput, INFINITE) != WAIT_OBJECT_0) //mutex IO scorp.
    {
        cout << "SCORPION: Erreur obtention mutex" << endl;
        CloseHandle(hMutexOutput);
        system("pause");
        exit(1);
    }
    setBasePosition(true); //initialiser position robot
    if (!ReleaseMutex(hMutexOutput) ) //libération mutex IO scorpion
    {
        cout << "SCORPION: Erreur liberation mutex" << endl;
        CloseHandle(hMutexOutput);
        system("pause");
        exit(1);
    }
}


/*-------------------------------------------------------------------------
  DESTRUCTEUR - FERMER ACCES ROBOT + FERMER MUTEX
 -------------------------------------------------------------------------*/
Scorpion::~Scorpion()
{
    dxl_terminate();
    CloseHandle(hMutexOutput);
}
```

```
/*-------------------------------------------------------------------------
  SETTER / GETTER VITESSE
 -------------------------------------------------------------------------*/
void Scorpion::setSpeed(int newSpeed)
{
    _speed = newSpeed;
}

int Scorpion::getSpeed()
{
    return _speed;
}



/*-------------------------------------------------------------------------
  PLACER ROBOT EN POSITION INITIALE
 -------------------------------------------------------------------------*/
void Scorpion::setBasePosition(bool wait)
{
    /* queue */
    moveMotor(TAIL_BASE, 680, 0);
    moveMotor(TAIL_MID, 760, 0);
    moveMotor(TAIL_END, 720, 0);
    if (wait)
        Sleep(250);

    /* rotation pattes */
    moveMotor(LEG_L1_ROTATE, 390, 0);
    moveMotor(LEG_R1_ROTATE, 630, 0);
    moveMotor(LEG_L2_ROTATE, 510, 0);
    moveMotor(LEG_R2_ROTATE, 510, 0);
    moveMotor(LEG_L3_ROTATE, 630, 0);
    moveMotor(LEG_R3_ROTATE, 390, 0);

    /* pli pattes */
    moveMotor(LEG_L3_BEND, 520, 0);
    moveMotor(LEG_R3_BEND, 500, 0);
    moveMotor(LEG_L2_BEND, 520, 0);
    moveMotor(LEG_R2_BEND, 500, 0);
    moveMotor(LEG_L1_BEND, 520, 0);
    moveMotor(LEG_R1_BEND, 500, 0);
}



/*-------------------------------------------------------------------------
  DEPLACEMENT D'UN MOTEUR
 -------------------------------------------------------------------------*/
void Scorpion::moveMotor(int motor, int position, int wait)
{
    /* déplacer moteur */
    dxl_write_word(motor, P_PRESENT_SPEED_L, getSpeed()); //vitesse
    dxl_write_word(motor, P_GOAL_POSITION_L, position);   //position

    /* attente optionnelle */
    if (wait)
        Sleep(wait);
}
```

7 - Conclusion

Le robot Bioloid est un dispositif très instructif qui offre beaucoup de possibilités. Un kit peut être assemblé de beaucoup de manières différentes, ce qui permet d'envisager une grande quantité d'usages. Par ailleurs, de nombreux types de programmation sont disponibles, ce qui rend le robot utilisable aussi bien par des informaticiens que des automaticiens. Cela permet également d'expérimenter différentes méthodes.

Les entrées et sorties du robot s'utilisent d'une façon radicalement différente du PETRA. Il n'est plus question ici de lire ou écrire l'ensemble des données mais plutôt de traiter chaque servomoteur au cas par cas. De plus, chaque moteur présente ici une plage de valeurs de position très importante, ainsi que d'autres caractéristiques (telles que la vitesse).

Il nous a été impossible de faire fonctionner simultanément les librairies *Dynamixel SDK* et *ZigBee SDK*. Pour cette raison, nous n'avons pas pu utiliser la manette pour contrôler le robot. Nous avons donc opté pour un contrôle au clavier. Vu que l'ordinateur devait alors rester connecté au robot, nous n'avons pas injecté l'application dans le microcontrôleur du robot. A la place, nous avons simplement choisi d'exécuter l'application sur l'ordinateur, afin de commander les servomoteurs du robot depuis l'ordinateur. Cette solution présente l'avantage de rendre les tests plus faciles et plus rapides à réaliser (l'injection du code dans le robot nécessite bon nombre d'étapes supplémentaires pour chaque test).

Avec un peu plus de temps, il aurait été intéressant d'explorer plus de possibilités et de trouver un moyen d'utiliser simultanément *Dynamixel SDK* et *ZigBee SDK*. Toutefois, notre but premier est atteint : contrôler le robot pour réaliser une série d'actions et de séquences. Ce projet nous a déjà beaucoup appris, tant au niveau de la programmation sur Windows qu'au niveau robotique.