

Miguel Hernando

**2018**

**Puerto entre APOLO , MATLAB y Python**

Este manual describe brevemente el puerto de comunicación TCP/IP que permite el control de Apolo desde otro programa. Además del protocolo básico se explican los distintos módulos de Matlab que implementan estas funcionalidades, así como la clase de Python con esta misma funcionalidad.

**Universidad Politécnica de Madrid**

**Centro de Automática y Robótica**

**ETSIDI**

**miguel.hernando@upm.es**

Contenido

[PROTOCOLO 3](#_Toc528678007)

[COMANDOS DISPONIBLES 5](#_Toc528678008)

[apoloCheckRobotJoints 5](#_Toc528678009)

[apoloGetLocation 6](#_Toc528678010)

[apoloGetLocationMRobot 7](#_Toc528678011)

[apoloMoveMRobot 8](#_Toc528678012)

[apoloPlace 9](#_Toc528678013)

[apoloPlaceMRobot 10](#_Toc528678014)

[apoloPlaceXYZ 11](#_Toc528678015)

[apoloSetRobotJoints 12](#_Toc528678016)

[apoloUpdate 13](#_Toc528678017)

[apoloGetAllultrasonicSensors 14](#_Toc528678018)

[apoloGetLaserData 15](#_Toc528678019)

[apoloGetLaserLandMarks 16](#_Toc528678020)

[apoloGetOdometry 17](#_Toc528678021)

[apoloGetUltrasonicSensor 18](#_Toc528678022)

[apoloResetOdometry 19](#_Toc528678023)

[ANEXO: 20](#_Toc528678024)

[ApoloMessage.h 20](#_Toc528678025)

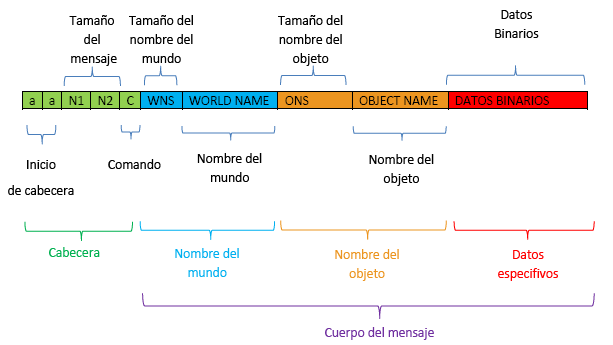
[ApoloMessage.cpp 22](#_Toc528678026)

# PROTOCOLO

Apolo dispone de un puerto de comunicación externo por medio de una conexión TCP/IP, de forma que cualquier sistema capaz de enviar y recibir datos binarios a través de dicho puerto de comunicación, podrá solicitar que se ejecuten comandos en el simulador.

El puerto de comunicaciones utilizado es el **12000**.

Los mensajes aceptados y respondidos por Apolo tienen el formato siguiente:



De estos campos, los únicos que aparecerá en todos los posibles mensajes son los relativos a la cabecera. El resto dependerán del tipo de comando que se transmita o del tipo de dato de respuesta. En este manual una cabecera de un comando determinado la indicaremos como HEADER('caracter del comando').

En general todas las **cadenas de caracteres** (nombres) se codifican de la misma forma. El primer carácter indica el tamaño (0 si no se especifica una cadena) , y en caso de que el tamaño sea mayor que cero, a continuación aparecerá la cadena en formato C. El tamaño del nombre incluye el cero de finalización. En este manual, a esta secuencia la denominaremos STRING(nombre)

Los **vectores de doubles**, se codifican siempre con un primer número entero que indica el tamaño y codificado con dos bytes (el primero es la parte menos significativa, y el segundo la más significativa (little endian)), y a continuación se transmiten los 64 bits del estandar de números reales, por orden creciente de cada elemento del vector. En el manual, a esta secuencia la denominaremos V\_DOUBLES(num,vector)

**Inicio de cabecera:** Bytes utilizados para distinguir el comienzo de un mensaje. Todos empezaran por los caracteres ‘a’ ‘a’.

**Tamaño del mensaje:** N1 indicara el numero de caracteres de 0 a 255, puesto que es el valor máximo que puede adoptar un byte, en caso de que el tamaño del mensaje sea superior a este, se utilizara el byte N2, ampliando el rango a 65536.

**Comando:** Byte que indicara el tipo de instrucción que contiene el mensaje, para que el receptor sepa como interpretar el contenido de este.

**Tamaño del nombre del mundo:** Byte que indica el número de caracteres de los que consta el nombre del mundo.

**Nombre del mundo:** Conjunto de bytes que contienen los caracteres que forman el nombre del mundo, el último carácter es un 0

**Tamaño del nombre del objeto:** Byte que indica el número de caracteres de los que consta el nombre del objeto.

**Nombre del objeto:** Byte que indica el número de caracteres de los que consta el nombre del objeto., el ultimo carácter es un cero.

**Datos binarios:** datos característicos de cada mensaje para la manipulación del objeto en cuestión.



**Ejemplo:**

El tamaño 46 se descompone en:

5 bytes de cabecera

8+1 bytes para el nombre del mundo

14+1 bytes para el nombre del robot

1 byte para el indicador de numero de articulaciones

2x8 bytes para codificar los dos doubles

# COMANDOS DISPONIBLES

## apoloCheckRobotJoints

**funcionalidad:**

Esta función permite mover las articulaciones de un robot articular y chequear si en la configuración resultante el robot colisiona con los objetos del entorno.

**Prototipo MatLab**: ret=apoloCheckRobotJoints(robot,values,world)

**Metodo Python**: def checkRobotJoints(self,robot, values, world='')

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Puma560'
* *values*: vector/lista de números reales. Su rango determinará cuantos ejes se intentarán establecer.
* *world*: mundo al que pertenece el robot que se quiere mover. En caso de omitirse, se moverá el primer robot que se encuentre bajo el nombre indicado si se recorren en orden creciente los mundos abiertos en Apolo.

**valor de retorno:**

***1/True -*** en caso de que el robot colisione con el entorno

***0/False-*** si la configuración es libre de colisión

**Ejemplo en Matlab**:

res=apoloCheckRobotJoints('Puma560',[0.5 0.23 -0.1],'world1');

if(res==1)

display('colisiona');

else

display('no colisiona');

end

**Ejemplo en Python:**

from Apolo import \*

ap=Apolo()

ap.checkRobotJoints("Puma 560",[0.5, 0.23, -0.1])

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('j')+STRING(WORLD)+STRING(ROBOT)+DATOS\_BINARIOS

**DATOS BINARIOS**: BYTE(n=tamaño vector)+n\*(DOUBLE(VALOR[i])

**información trama de respuesta:**

Para el valor true: HEADER con el COMANDO 'T'

Para el valor false: HEADER con el COMANDO 'F'

## apoloGetLocation

**funcionalidad:**

Esta función obtiene la localización espacial de un objeto (habitualmente un robot).

**Prototipo MatLab**: ret=apoloGetLocation(robot,world)

**Metodo Python**: def getLocation(self, object, world='')

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot u el objeto e.g: 'Puma560'
* *world*: mundo al que pertenece el elemento que se quiere mover. Omisible.

**valor de retorno:**

***Double [6]/Lista de reales -*** retorna un vector/lista de 6 componentes. Las tres primeras corresponden a la posición, mientras que las tres últimas a los ángulos roll, pitch y yaw en radianes.

**Ejemplo Matlab**:

>> apoloGetLocation('pieza')

ans =

-0.2500 0.2000 0.0600 1.5708 0.6912 0.9425

**Ejemplo Python:**

print(ap.getLocation('Puma 560'))

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('G')+STRING(WORLD)+STRING(OBJECT)

**DATOS BINARIOS**: No procede

**información trama de respuesta:**

HEADER ('T')+ V\_DOUBLES(6,posicion y orientacion)

## apoloGetLocationMRobot

**funcionalidad:**

Esta función obtiene la posición de un vehículo con ruedas

**Prototipo MatLab**: ret=apoloGetLocationMRobot(robot,world)

**Metodo Python**:

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Neo'
* *world*: mundo al que pertenece el elemento que se quiere mover. Omisible.

**valor de retorno:**

***Double [4] -*** retorna un vector de 4 componentes. Las tres primeras corresponden a la posición, mientras que la última es el ángulo de rotación en Z..

**Ejemplo Matlab**:

>> apoloGetLocationMRobot('Pioneer3AT')

ans =

0.7000 2.5000 0 0

**Ejemplo Python:**

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('g')+STRING(WORLD)+STRING(ROBOT)

**DATOS BINARIOS**:

**información trama de respuesta:**

HEADER ('D')+ V\_DOUBLES(4,[x y z rz])

## apoloMoveMRobot

**funcionalidad:**

Esta mueve un robot movil a la velocidad indicada durante un tiempo tambien establecido. El robot aplicará las restricciones cinemáticas y geométricas. Por tanto si colisionase, se quedaría parado en la última posición sin colisión.

**prototipo**:

function ret = apoloMoveMRobot(robot,speeds,time,world)

def moveWheeledBase(self, robot, speed, rotspeed, time, world='')

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Neo'
* *speeds*: vector de dos doubles que son las consignas de velocidad (normalmente avance y rotación)
* *time*: tiempo en segundos que durará la acción. Conviene indicar pasos pequeños para evitar que el robot se "salte" o cheque contra objetos por razones de discretización de la trayectoria.
* *world*: cadena de caracteres indicación del mundo. Omitible

**valor de retorno:**

***1/True -*** en caso de que el robot se haya movido correctamente

***0/False-*** si el robot ha chocado o carece de suelo para moverse

**Ejemplo**:

>> apoloMoveMRobot('Pioneer3AT',[0.1 0.0],0.1)

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('m')+STRING(WORLD)+STRING(ROBOT)+DATOS\_BINARIOS

**DATOS BINARIOS**: DOUBLE(speed1)+DOUBLE(speed2)+DOUBLE(time)

**información trama de respuesta:**

Para el valor true: HEADER con el COMANDO 'T'

Para el valor false: HEADER con el COMANDO 'F'

## apoloPlace

**funcionalidad:**

Posiciona un objeto en la posición y orientación indicada independientemente de si colisiona o no con el entorno.

**prototipo**:

function apoloPlace (object,pos,or,world)

**parámetros**:

* *object*: cadena de caracteres que identifica el elemento a mover e.g: 'pieza'
* *pos: vector de 3 doubles que indican la posición*
* *or: vector de 3 doubles que indican la orientación en radianes (rpy)*
* *world*: cadena de caracteres indicación del mundo. Omisible

**retorno**:

**Ejemplo**:

>> apoloPlace ('pieza',[1 1 1],[0.5 0 0])

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('P')+STRING(WORLD)+STRING(OBJECT)+DATOS\_BINARIOS

**DATOS BINARIOS**: DOUBLE(x)+DOUBLE(y)+DOUBLE(z)+(DOUBLE(rx)+DOUBLE(ry)+DOUBLE(rz)

## apoloPlaceMRobot

**funcionalidad:**

intenta posicionar un robot movil en la posición x y z , y la orientación dada por el angulo. Para ello *deja caer* desde la altura indicada el robot y comprueba si la posición de caida es valida por el número de apoyos y porque el cuerpo del robot no colisione.

**prototipo**:

function ret=apoloPlaceMRobot(robot,pose,angle,world)

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Neo'
* *pose*: vector de tres doubles que son las coordenadas de posición
* *angle*: angulo en radianes de rotación en Z
* *world*: cadena de caracteres indicación del mundo. Omisible

**retorno**:

* *1, en caso de tener éxito, 0 en caso contrario.*

**Ejemplo**:

>> apoloPlaceMRobot('Pioneer3AT',[0.7 0.7 0],0.23)

ans =

1

>>

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('p')+STRING(WORLD)+STRING(ROBOT)+DATOS\_BINARIOS

**DATOS BINARIOS**: DOUBLE(x)+(DOUBLE(y)+DOUBLE(z)+DOUBLE(rz)

## apoloPlaceXYZ

**funcionalidad:**

Posiciona un objeto en las coordenadas XYZ independientemente de si colisiona o no con el entorno..

**prototipo**:

function apoloPlaceXYZ(object,x,y,z,world)

**parámetros**:

* *object*: cadena de caracteres que identifica el elemento a mover e.g: 'pieza'
* *x,y,z: doubles que indican la posición*
* *world*: cadena de caracteres indicación del mundo. Omisible

**retorno**:

**Ejemplo**:

>> apoloPlaceXYZ('Pioneer3AT',1,1,1)

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: la misma que apoloPlace pero con rx=ry=rz=0

**DATOS BINARIOS**:

## apoloSetRobotJoints

**funcionalidad:**

Esta función permite mover las articulaciones de un robot articular, pero no valida si colisiona o no.

**prototipo**:

function apoloSetRobotJoints(robot,values,world)

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Puma560'
* *values*: vector de números reales. Su rango determinará cuantos ejes se intentarán establecer.
* *world*: mundo al que pertenece el robot que se quiere mover. En caso de omitirse, se moverá el primer robot que se encuentre bajo el nombre indicado si se recorren en orden creciente los mundos abiertos en Apolo.

**Ejemplo**:

>>apoloSetRobotJoints('Puma560',[0.5 0.23 -0.1],'world1');

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('J')+STRING(WORLD)+STRING(ROBOT)+DATOS\_BINARIOS

**DATOS BINARIOS**: BYTE(n=tamaño vector)+n\*(DOUBLE(VALOR[i])

## apoloUpdate

**funcionalidad:**

Actualiza la visualización de un mundo en Apolo. En caso de omitirse World, actualizará todos los mundos cargados en Apolo.

**prototipo**:

function apoloUpdate(world)

**parámetros**:

* *world*: cadena de caracteres indicación del mundo. Omisible

**retorno**:

**Ejemplo**:

>> apoloUpdate('World1')

>> apoloUpdate

>>

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('U')+STRING(World)

## apoloGetAllultrasonicSensors

**funcionalidad:**

Esta función obtiene por orden de definición todos los ultrasonidos dependientes de un objeto (en primer nivel). Es decir, no obtendrá los ultrasonidos que dependen de un objeto dependiente del indicado.

**Prototipo MatLab**: ret=apoloGetAllultrasonicSensors(object,world)

**Metodo Python**: def getDependentUSensors(self, object, world=**''**)

**parámetros**:

* *object*: cadena de caracteres que identifica el objeto base e.g: 'Pioneer'
* *world*: mundo al que pertenece el objeto. En caso de omitirse, se considerará el primer objeto que se encuentre bajo el nombre indicado si se recorren en orden creciente los mundos abiertos en Apolo.

**valor de retorno:**

***Vector de doubles.*** *En Matlab es una matriz, en Python es una lista. En ambos casos son las medidas de distancia de los sensores en el orden de definición respecto del objeto.*

**Ejemplo en Matlab**:

>> a = apoloGetAllultrasonicSensors('Marvin')

**Ejemplo en Python:**

>>> a = ap.getDependentUSensors('Marvin')

[3.0, 3.0, 0.5851024075142821]

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('d')+STRING(WORLD)+STRING(OBJETO)

**información trama de respuesta:**

HEADER ('D')+ V\_DOUBLES(n,[…]) siendo n el número de sensores encontrados

## apoloGetLaserData

**funcionalidad:**

Esta función obtiene los valores en distancia del conjunto de haces laser de un barrido del sensor. Es necesario por tanto conocer las características del mismo (ángulo, paso, y número de medidas).

**Prototipo MatLab**: ret=apoloGetLaserData(laser,world)

**Metodo Python**: def getLaserData(self, laser, world='')

**parámetros**:

* *laser*: cadena de caracteres que identifica el sensor laser e.g: 'LM 100'
* *world*: mundo al que pertenece el laser que se quiere consultar.

**valor de retorno:**

***Vector Doubles –*** retorna un vector/matriz de medidas del laser. Desde la medida del ángulo más pequeño hasta la del mayor.

**Ejemplo en Matlab**:

**Ejemplo en Python:**

>>> values = ap.getLaserData('LMS100')

>>> type(values)

<class 'list'>

>>> len (values)

541

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('l')+STRING(WORLD)+STRING(LASER)

**información trama de respuesta:**

HEADER ('D')+ V\_DOUBLES(n,[…]) siendo n el número de medidas

## apoloGetLaserLandMarks

**funcionalidad:**

Esta función obtiene las medidas de distancia y ángulo de los “LandMArks” visibles por el laser. Retorna por tanto una lista de elementos estructurados. Cada elemento de esta lista contiene una tupla/estructura con el identificador, el ángulo en radianes y la distancia.

**Prototipo MatLab**: ret=apoloGetLaserLandMarks(laser,world)

**Metodo Python**: def getLaserLandMarks(self, laser, world='')

**parámetros**:

* *laser*: cadena de caracteres que identifica el laser e.g: 'myLMS100'
* *world*: mundo al que pertenece el laser que se quiere consultar. En caso de omitirse se procede como siempre.

**valor de retorno:**

***MATLAB –*** *estructura con 3 campos, cada uno un vector de N componentes, correspondiente a las N balizas vistas por el laser:* id ,angle , distance

***PYTHON-*** *Lista de N tuplas [(ID, angle, distance), …]*

**Ejemplo en Matlab**:

>> apoloGetLaserLandMarks('LMS100')

ans =

struct with fields:

id: [1 2]

angle: [0.4948 -1.7708]

distance: [5.6069 2.8204]

**Ejemplo en Python:**

>>> from apolo import \*

>>> ap = Apolo()

>>> ap.getLaserLandMarks('LMS100')

[(1, 0.5127125609331823, 5.5969096536351275),

(2, -1.7431719949660114, 2.8617808350918064)]

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('B')+STRING(WORLD)+STRING(LASER)

**información trama de respuesta:**

**TRAMA**: HEADER ('b')+ VECTOR(n: INT16,[INT16, DOUBLE, DOUBLE]) siendo n el número de medidas

## apoloGetOdometry

**funcionalidad:**

Obtiene el valor de odometría medido por un robot con ruedas.

**Prototipo MatLab**: ret=apoloGetOdometry(robot, world)

**Metodo Python**: def getOdometry(self, robot, world='')

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: 'Marvin'
* *world*: mundo al que pertenece el robot.

**valor de retorno:**

*Vector de tres doubles [x, y , theta] que indican la posición en metros hasta entonces computada por el robot desde el último reset o inicio. Queda afectada por ruido gaussiano en caso de estar este definido para el robot en concreto.*

**Ejemplo en Matlab**:

pos=apoloGetOdometry(robot)

A=[pos]

for i = 1:n

apoloMoveMRobot(robot, [0.05, 0.05], 0.1);

apoloUpdate()

a=apoloGetOdometry(robot);

A=[A ; a];

end

**Ejemplo en Python:**

>>> ap.getOdometry('Marvin')

[1.590259967336959, 2.3041701016169363, 2.3505950006518024]

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('o')+STRING(WORLD)+STRING(ROBOT)

**información trama de respuesta:**

HEADER ('D')+ V\_DOUBLES(3,[x y theta])

## apoloGetUltrasonicSensor

**funcionalidad:**

Esta función permite obtener el valor de un sensor de ultrasonido identificable por su nombre.

**Prototipo MatLab**: ret=apoloGetUltrasonicSensor(sensor,world)

**Metodo Python**: def getUSensor(self, sensor, world='')

**parámetros**:

* *sensor*: cadena de caracteres que identifica el sensor e.g: ‘ur0’
* *world*: mundo al que pertenece el sensor.

**valor de retorno:**

Un valor real que es la distancia medida por el sensor

**Ejemplo en Matlab**:

res=apoloGetUltrasonicSensor ('ur0');

**Ejemplo en Python:**

A = ap.getUSensor("ur0”)

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('u')+STRING(WORLD)+STRING(SENSOR)

**información trama de respuesta:**

HEADER ('D')+ V\_DOUBLES(1,[d])

## apoloResetOdometry

**funcionalidad:**

Esta función permite resetear la odometría del robot. Pondrá a cero la referencia interna y adoptará los valores indicados si los hubiera como offset inicial. Si no se indica nada, se considerará la situación actual el nuevo cero.

**Prototipo MatLab**: apoloResetOdometry(robot,pose,world)

**Metodo Python**: def resetOdometry(self, robot, pose2d=(0, 0, 0), world='')

**parámetros**:

* *robot*: cadena de caracteres que identifica el robot e.g: ‘Marvin’
* *pose*: vector/lista de números reales [x y theta]. Valor por omisión [0 0 0]
* *world*: mundo al que pertenece el robot. En caso de omitirse se buscara el primer robot entre los mundos cargados en apolo.

**Ejemplo en Matlab**:

apoloResetOdometry ('Marvin',[0.5 0.23 0]);

**Ejemplo en Python:**

ap.resetOdometry(‘Marvin’)

**Información de la trama TCP/IP COMANDO:**

**TRAMA**: HEADER('R')+STRING(WORLD)+STRING(ROBOT)+DATOS\_BINARIOS

**DATOS BINARIOS**: DOUBLE(X)+DOUBLE(Y)+DOUBLE(THETA)

# ANEXO:

## ApoloMessage.h

#pragma once

#include "mrcore.h"

#define AP\_NONE 0

#define AP\_GET\_LASER\_LM 'B'

#define AP\_DVECTOR 'D'

#define AP\_FALSE 'F'

#define AP\_GETLOCATION 'G'

#define AP\_SETJOINTS 'J'

#define AP\_LINK\_TO\_ROBOT\_TCP 'L'

#define AP\_PLACE 'P'

#define AP\_RESET\_ODOMETRY 'R'

#define AP\_TRUE 'T'

#define AP\_UPDATEWORLD 'U'

#define AP\_LM\_INFO 'b'

#define AP\_GET\_DEP\_USENSORS 'd'

#define AP\_GETLOCATION\_WB 'g'

#define AP\_CHECKJOINTS 'j'

#define AP\_GET\_LASER\_DATA 'l'

#define AP\_MOVE\_WB 'm'

#define AP\_GET\_WB\_ODOMETRY 'o'

#define AP\_PLACE\_WB 'p'

#define AP\_GET\_USENSOR 'u'

#include <vector>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*This class implements the protocol for easily connect to apolo

An apolo message pointers to an external buffer. Is simply an interpreter

of raw data. Therefore, use have to be carefull.

\*/

class ApoloMessage

{

char \*pData;//pointer to a byte secuence that has a message (header+size+type+specific data)

char \*world,\*name,\*bindata; //utility fields to avoid reinterpretation

int size;

char type;

ApoloMessage(char \*buffer,int size,char type);

public:

static int writeSetRobotJoints(char \*buffer, char \*world, char \*robot, int num, double \*values);

static int writeCheckColision(char \*buffer, char \*world, char \*robot, int num, double \*values);

static int writeUpdateWorld(char \*buffer, char \*world);

static int writeBOOL(char \*buffer, bool val);

static int writePlaceObject(char \*buffer, char \*world,char \*object, double \*xyzrpy);

static int writePlaceWheeledBase(char \*buffer, char \*world,char \*robot, double \*xyzy);

static int writeMoveWheeledBase(char \*buffer, char \*world,char \*robot, double \*sp\_rs\_t); //speed,rot\_speed,time

static int writeGetLocation(char \*buffer, char \*world,char \*object);

static int writeGetLocationWheeledBase(char \*buffer, char \*world,char \*robot);

static int writeDoubleVector(char \*buffer, int num, double \*d);

static int writeDoubleVector(char \*buffer, std::vector<double> v);

static int writeLinkToRobotTCP(char \*buffer, char \*world,char \*robot,char \*object);

static int writeGetLaserData(char \*buffer, char \*world, char \*laser);

static int writeGetOdometry(char \*buffer, char \*world, char \*robot); //last x, last y, last yaw(rad), noise

static int writeGetUltrasonicSensor(char \*buffer, char \*world, char\* name); //AP\_GET\_USENSOR

static int writeGetDependentUltrasonicSensors(char \*buffer, char \*world, char\* object);// AP\_GET\_DEP\_USENSORS

static int writeGetLaserLandMarks(char \*buffer, char \*world, char \*laser); //AP\_GET\_LASER\_LM

static int writeLandMarkInfoVector(char \*buffer, std::vector<mr::LaserSensorSim::LandMarkInfo> &v); //AP\_LM\_INFO

static int writeResetOdometry(char \*buffer, char \*world, char \*robot, double \*xyt); //AP\_RESET\_ODOMETRY

static ApoloMessage \*getApoloMessage(char \*\*buffer, int max);

char \*getWorld(){return world;}

char \*getObjectName(){return name;}

char getType(){return type;}

int getSize(){return size;}

int getIntAt(int offset);

int getUInt16At(int offset);

double getDoubleAt(int offset);

char getCharAt(int offset);

char \*getStringAt(int offset);

};

## ApoloMessage.cpp

#include "apoloMessage.h"

#include <string.h>

/\*\*UTILITY UNIONS FOR CONVERSIONS\*\*/

union double2byteConversor

{

char bytes[8];

double real;

};

union int2byteConversor

{

char bytes[4];

int integer;

};

typedef unsigned char uchar;

/\*\*

writes intp buffer the message for moving a robot in a worl

if world=null... is equivalent to any.

returns the message size

\*\*/

inline int Apolo\_writeString(char \*buffer, char \*cad){

int n=0,len;

if(cad!=0){ //not null

if(cad[0]==0)buffer[n++]=0;//empty string

else{

len=1+(uchar)strlen(cad);

((uchar \*)buffer)[n++]=(uchar)((len>255)?255:len);

for(int i=0;i<len-1;i++)buffer[n++]=cad[i];

buffer[n++]=0;

}

}else buffer[n++]=0;

return n;

}

inline int Apolo\_writeDouble(char \*buffer, double val){

double2byteConversor aux;

aux.real=val;

for(int i=0;i<8;i++)buffer[i]=aux.bytes[i];

return 8;

}

inline int Apolo\_writeUInt16(char \*message, int &num)

{

if(num>65535)num=65535;

if(num<0)num=0;

((uchar \*)message)[0]=(uchar)(num%256);

((uchar \*)message)[1]=(uchar)(num/256);

return 2;

}

inline void Apolo\_insertSize(char \*message, int size)//size including the header

{

((uchar \*)message)[2]=(uchar)(size%256);

((uchar \*)message)[3]=(uchar)(size/256);

}

//tamaño minimo de mensaje es 5

inline int Apolo\_writeHeader(char\*buffer,char command) //escribe la cabecera

{

int n=0;

buffer[0]='a';

buffer[1]='a';

Apolo\_insertSize(buffer,5);

buffer[4]=command;

return 5;

}

int ApoloMessage::writeSetRobotJoints(char \*buffer, char \*world, char \*robot, int num, double \*values)

{

int n=0;

n+= Apolo\_writeHeader(buffer,AP\_SETJOINTS);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,robot);//robot

if(num<0)num=0;

if(num>255)num=255;

((uchar \*)buffer)[n++]=(uchar)num;//num joints

for(int i=0;i<num;i++)

n+= Apolo\_writeDouble(buffer+n,values[i]);

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writePlaceObject(char \*buffer, char \*world,char \*object, double \*xyzrpy)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_PLACE);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,object);//object

for(i=0;i<6;i++)

n+= Apolo\_writeDouble(buffer+n,xyzrpy[i]);

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeMoveWheeledBase(char \*buffer, char \*world,char \*robot, double \*sp\_rs\_t)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_MOVE\_WB);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,robot);//robot

for(i=0;i<3;i++)//speed, rot speed, time

n+= Apolo\_writeDouble(buffer+n,sp\_rs\_t[i]);

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writePlaceWheeledBase(char \*buffer, char \*world,char \*robot, double \*xyzy)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_PLACE\_WB);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,robot);//robot

for(i=0;i<4;i++)//x,y,z, rot z

n+= Apolo\_writeDouble(buffer+n,xyzy[i]);

Apolo\_insertSize(buffer,n);

return n;

}

//the same message But changes the command id

int ApoloMessage::writeCheckColision(char \*buffer, char \*world, char \*robot, int num, double \*values)

{

int n=writeSetRobotJoints(buffer,world,robot,num,values);

buffer[4]=AP\_CHECKJOINTS;

return n;

}

int ApoloMessage::writeGetLocation(char \*buffer, char \*world,char \*object)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_GETLOCATION);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,object);//robot

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeGetLocationWheeledBase(char \*buffer, char \*world,char \*robot)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_GETLOCATION\_WB);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,robot);//robot

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeUpdateWorld(char \*buffer, char \*world)

{

int n=0;

n+= Apolo\_writeHeader(buffer,AP\_UPDATEWORLD);//command

n+= Apolo\_writeString(buffer+n,world);//world

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeLinkToRobotTCP(char \*buffer, char \*world,char \*robot,char \*object)

{

int n=0,i;

n+= Apolo\_writeHeader(buffer,AP\_LINK\_TO\_ROBOT\_TCP);//command

n+= Apolo\_writeString(buffer+n,world);//world

n+= Apolo\_writeString(buffer+n,robot);//robot

n+= Apolo\_writeString(buffer+n,object);//robot

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeGetLaserData(char \*buffer, char \*world, char \*laser)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_GET\_LASER\_DATA);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, laser);//laser

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeGetLaserLandMarks(char \*buffer, char \*world, char \*laser)//AP\_GET\_LASER\_LM

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_GET\_LASER\_LM);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, laser);//laser

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeGetOdometry(char \*buffer, char \*world, char \*robot)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_GET\_WB\_ODOMETRY);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, robot);//robot

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeGetUltrasonicSensor(char \*buffer, char \*world, char\* name)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_GET\_USENSOR);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, name);//laser

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeGetDependentUltrasonicSensors(char \*buffer, char \*world, char\* object)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_GET\_DEP\_USENSORS);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, object);//laser

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeResetOdometry(char \*buffer, char \*world, char \*robot, double \*xyt) //AP\_RESET\_ODOMETRY

{

int n = 0, i;

n += Apolo\_writeHeader(buffer, AP\_RESET\_ODOMETRY);//command

n += Apolo\_writeString(buffer + n, world);//world

n += Apolo\_writeString(buffer + n, robot);//robot

for (i = 0; i<3; i++)//x,y, rot z

n += Apolo\_writeDouble(buffer + n, xyt[i]);

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeDoubleVector(char \*buffer, int num, double \*d)

{

int n=0;

n+= Apolo\_writeHeader(buffer,AP\_DVECTOR);//command

n+= Apolo\_writeUInt16(buffer+n,num);

for(int i=0;i<num;i++)

n+= Apolo\_writeDouble(buffer+n,d[i]);

Apolo\_insertSize(buffer,n);

return n;

}

int ApoloMessage::writeDoubleVector(char \*buffer, std::vector<double> v)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_DVECTOR);//command

int num = (int)v.size();

n += Apolo\_writeUInt16(buffer + n, num);

for (int i=0; i<num; ++i)

n += Apolo\_writeDouble(buffer + n, v[i]);

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeLandMarkInfoVector(char \*buffer, std::vector<mr::LaserSensorSim::LandMarkInfo> &v)

{

int n = 0;

n += Apolo\_writeHeader(buffer, AP\_LM\_INFO);//command

int num = (int)v.size();

n += Apolo\_writeUInt16(buffer + n, num);

for (int i = 0; i < num; ++i) {

n += Apolo\_writeUInt16(buffer + n, v[i].ID);

n += Apolo\_writeDouble(buffer + n, v[i].ang);

n += Apolo\_writeDouble(buffer + n, v[i].dist);

}

Apolo\_insertSize(buffer, n);

return n;

}

int ApoloMessage::writeBOOL(char \*buffer, bool val)

{

int n=0;

char command=AP\_FALSE;

if(val)command=AP\_TRUE;

n+= Apolo\_writeHeader(buffer,command);//command

Apolo\_insertSize(buffer,n);

return n;

}

ApoloMessage::ApoloMessage(char \*buffer,int size,char type)

{

char \*aux;

pData=buffer;

this->size=size;

this->type=type;

if(type==AP\_NONE)return;

world=bindata=name=0;

switch(type)

{//command with world and name

case AP\_SETJOINTS:

case AP\_CHECKJOINTS:

case AP\_PLACE:

case AP\_PLACE\_WB:

case AP\_MOVE\_WB:

case AP\_GETLOCATION\_WB:

case AP\_GETLOCATION:

case AP\_LINK\_TO\_ROBOT\_TCP:

case AP\_GET\_LASER\_DATA:

case AP\_GET\_WB\_ODOMETRY:

case AP\_GET\_USENSOR:

case AP\_GET\_DEP\_USENSORS:

case AP\_GET\_LASER\_LM:

case AP\_RESET\_ODOMETRY:

if(pData[5]!=0){

world=pData+6;

aux=world+((uchar \*)pData)[5];

}else aux=pData+6;

if(aux[0]!=0){

name=aux+1;

aux=name+((uchar \*)aux)[0];

}else aux++;

bindata=aux;

break;

case AP\_UPDATEWORLD://commands with world only

if(pData[5]!=0){

world=pData+6;

aux=world+((uchar \*)pData)[5];

}else aux=pData+6;

bindata=aux;

break;

default: //commands without world neither

bindata=pData+5;

break;

}

}

//se considera que el buffer contiene mensajes completos (pueden ser varios)... si son parciales, se desecharán

ApoloMessage \*ApoloMessage::getApoloMessage(char \*\*buffer, int max)

{

int i=0;

while(i+4<max){

if(((\*buffer)[i]=='a')&&((\*buffer)[i+1]=='a'))

{

int size=(((uchar \*)(\*buffer))[i+2])+(((uchar \*)(\*buffer))[i+3])\*255;

char type=(\*buffer)[i+4];

//si el mensaje es correcto... crea el mensaje y situa el puntero al final

//ojo... el mensaje mantiene unos punteros sobre el buffer original. El mensaje no reserva memoria

if(i+size<=max){

ApoloMessage \*message=new ApoloMessage((\*buffer)+i,size,type);

\*buffer=\*buffer+size;

return message;

}//si no lo es retorna null

else return 0;

}

i++;

}

return 0;

}

int ApoloMessage::getIntAt(int offset)

{

int2byteConversor aux;

if(offset+(bindata-pData)+4>size)return 0;

for(int i=0;i<4;i++)aux.bytes[i]=bindata[offset+i];

return aux.integer;

}

int ApoloMessage::getUInt16At(int offset)

{

if(offset+(bindata-pData)+2>size)return 0;

return (((uchar \*)(bindata))[offset])+(((uchar \*)(bindata))[offset+1])\*255;

}

double ApoloMessage::getDoubleAt(int offset)

{

double2byteConversor aux;

if(offset+(bindata-pData)+8>size)return 0;

for(int i=0;i<8;i++)aux.bytes[i]=bindata[offset+i];

return aux.real;

}

char ApoloMessage::getCharAt(int offset)

{

if(offset+(bindata-pData)+1>size)return 0;

return bindata[offset];

}

char \*ApoloMessage::getStringAt(int offset)

{

if(offset+(bindata-pData)+1>size)return 0;

uchar tam=((uchar \*)(bindata))[offset];

if(tam==0)return 0;

if(offset+(bindata-pData)+tam+1>size)return 0;

else return bindata+offset+1;

}