Khepera Simulator version 2.0
User Manual

Olivier MICHEL
E-mail: `om@alto.unice.fr`, Web: `http://wwwi3s.unice.fr/~om/`
University of Nice – Sophia Antipolis, Laboratoire I3S, CNRS
bât. 4, 250, av. A. Einstein 06560 Valbonne, France

March 1, 1996

1 License Agreement

Olivier Michel. *Khepera Simulator* Package version 2.0: Freeware mobile robot simulator written at the University of Nice Sophia–Antipolis by Olivier Michel. Downloadable from the World Wide Web at `http://wwwi3s.unice.fr/~om/khep-sim.html`

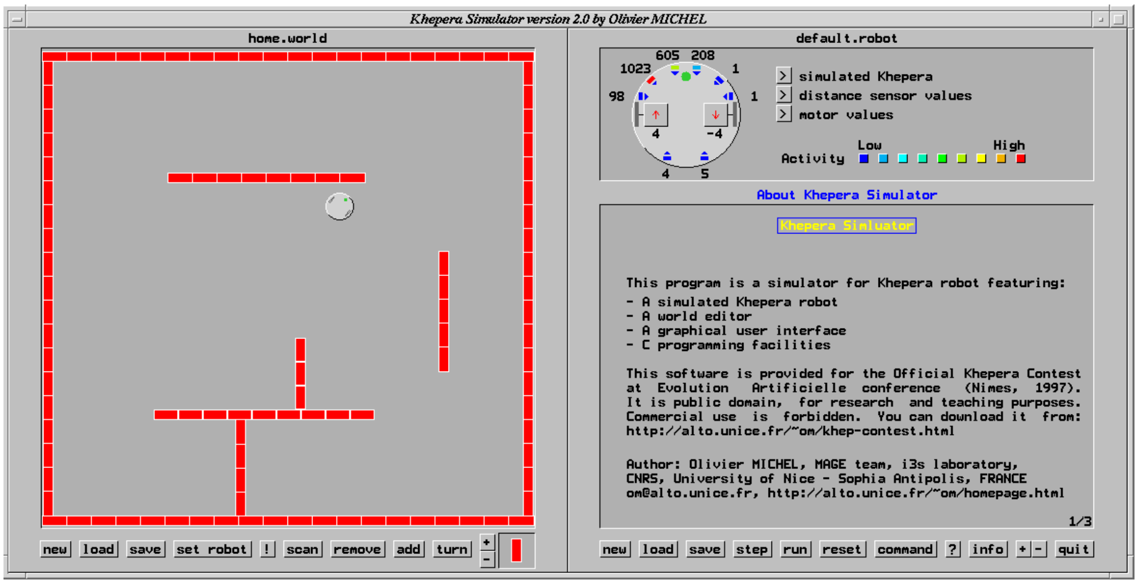


Figure 1: *Khepera Simulator*

2 Introduction

This package allows to write control algorithms (neural networks, classifier system, or anything else you may imagine) using C or C++ languages. A library of functions is provided that permits to drive the robot and display results. The simulator runs on Unix workstations and features a nice X11 colorful graphical interface. Some examples of controllers are given within the package, including a neural network controller. The simulator also features the ability to drive a real *Khepera* robot, so you can very easily transfer your simulation results to the real robot by clicking on a button.

The screen of the *Khepera Simulator* is divided into two parts: the “world” part stands on the left while the “robot” part stands on the right (see figure 1). In the world part, one can observe the behavior of the robot in its environment whereas in the robot part, one can observe what is going on inside the robot (sensors, motors and controller).

2.1 Description of the world

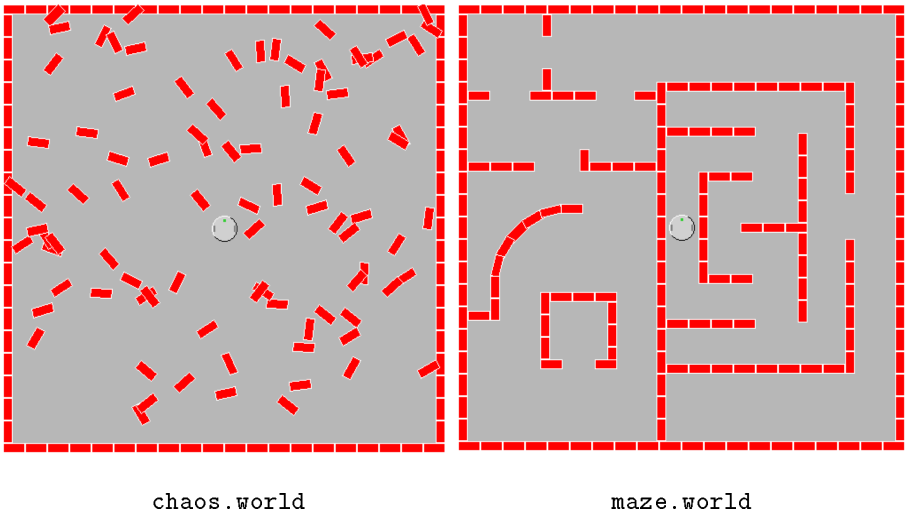


Figure 2: Two examples of simulated worlds

Various worlds for the robot are available in the `SIM/WORLD/` directory. Press the “`load`” button and type the world file name (without the `.world` extension) to load one of them. Moreover, it is possible to edit them or to design a new world from scratch by pressing the “`new`” button. Resulting worlds can be saved using the “`save`” button. Bricks, corks or lamps are laid in the environment resulting in more or less complex mazes (see for example figure 2). The real dimensions of this simulated environment (comparing to the real robot *Khepera*) are $1m \times 1m$.

To add an object, select it using the “`+`” and “`-`” buttons. If you want to turn an object on itself (for a brick for example), then press the “`turn`” button as many times as it is necessary. Then, to set the object at a location in the world, press the “`add`” button and drop the object in the world, then unpress the “`add`” button.. If you want to remove objects, press the “`remove`” button and click on the objects you want to remove. To get out of the remove mode, unpress the “`remove`” button. Once the bricks and corks have been laid, it is necessary to press the “`scan`” button before the robot could perceive them. It is possible to check what the robot can perceive by pressing the “`!`” button.

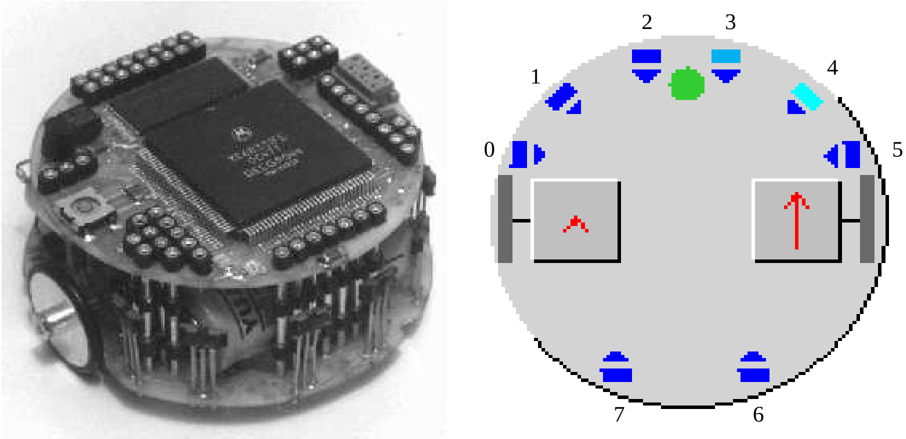


Figure 3: *Khepera* (5 cm diameter) and its simulated counterpart

2.2 Description of the robot

2.2.1 Presentation

Dedicated to *Khepera* [3], the simulated mobile robot includes 8 infrared sensors allowing it to detect by reflexion (small rectangles) the proximity of objects in front of it, behind it, and to the right and the left sides of it. Each sensor returns a value ranging between 0 and 1023 represented in color levels. 0 means that no object is perceived while 1023 means that an object is very close to the sensor (almost touching the sensor). Intermediate values may give an approximate idea of the distance between the sensor and the object. These sensors can also measure the level of ambient light (small triangles) all around the robot. They return a value displayed in color levels close to 500 in the dark and close to 50 in front of a light source. Each motor can take a speed value ranging between -10 and $+10$. Red arrows on the motors indicate this speed.

2.2.2 Motor Model

The model of the simulated motors are straight forward: the robot moves accordingly to the speed set by the the user. A random noise of $\pm 10\%$ is added to the amplitude of the motor speed while a random noise of $\pm 5\%$ is added to the direction resulting from the difference of the speeds of the motors (see function `SolveEffectors(...)` in source code file `robot.h` for more details).

2.2.3 Sensor Model

To calculate its distance value output, a simulated sensor explores a set of 15 points in a triangle in front of it. An output value is computed as a function of the presence (or the absence) of obstacles at these points. A random noise corresponding to $\pm 10\%$ of its amplitude is added to the distance value output (see function `IRSensorDistanceValue(...)` in source code file `robot.h` for more details).

The light value output is computed accordingly to the distance and the angle between the sensor and the light source. A $\pm 5\%$ noise is added to this value (see function `IRSensorLightValue(...)` in source code file `robot.h` for more details).

2.2.4 Operating the Robot

To set the robot at a given location in the world, press the button "`set robot`" (in the world part) and click somewhere in the world (possibly not on an object). The robot may also be oriented in the direction you like. To do this, press the "`command`" button and type for example `set angle 45`, you will see the robot turn to reach the 45 degrees position. Figure 3 shows the simulated robot at 90 degrees position. If it was looking to the right (resp. to the left), it would have been at 0 degrees (resp. 180 degrees). If you press the "`run`" button, `sim` will call continuously the user control function `RobotStep(robot)` (written in C in `user.c` source file) until the "`run`" is unpressed. If you want to observe a step by step run of the robot, then press the "`step`" button and the function `RobotStep(robot)` will be executed once. The "?" button allows to test the sensors of the robot (especially useful for the real robot).

3 Programming

3.1 Introduction

This section explains how to program your own *Khepera* controller. It describes the different files you need to know and starts up with a tutorial example. You will find in the appendix A a complete list of the C structures and functions necessary to program *Khepera Simulator*. Appendix B contains the directory structure of the package.

3.2 Simulator source files: do not modify them

The simulator source files are in the directory `SIM/SRC`. They must not be modified. This is very important for further updates of the software. These sources are written in ANSI C. So, if you want to write your controller in C, compile these sources with a C compiler (`gcc -c`), and if you prefer C++, you can also compile these sources with a C++ compiler (replace `gcc -c` by `g++ -c` in the file `makefile`), so that it will be easier to link them together (`gcc`) with your controller.

3.3 User files

3.3.1 Preferences: the .simrc file

The file `.simrc` is a preference file concerning hardware configurations. It is read by *Khepera Simulator* each time `sim` is executed. It's a hidden file in `/SIM` directory, so you need to type something like `ls -a` to see it. It contains 3 important parameters which you may edit:

- `KHEPERA_AVAILABLE:` may be `TRUE` or `FALSE`, depending if a real *Khepera* robot is connected or not.
- `SERIAL_PORT:` is serial port device to which the robot is connected (if available). It could be `/dev/ttya` standing for serial port A on a Sun workstation, but it generally depends on the kind of computer you use. This value is used only when `KHEPERA_AVAILABLE` is `TRUE`.
- `MONODISPLAY:` may be `TRUE` or `FALSE` according to the type of screen used. Setting it to `TRUE` allows to run *Khepera Simulator* on a monochrome display.

When calling `sim`, the option flag `-s` allows to run *Khepera Simulator* in "simulation only" mode (do not make use of the serial link for the real *Khepera*) even if `KHEPERA_AVAILABLE` is `TRUE`.

3.3.2 Controller source files

These files are yours. You can modify them the way you want. You can also add new files (and consequently modify the associated `makefile`). They are in `SIM/USER` directory. A version of the basic empty controller files is available in `SIM/EXAMPLES/EXAMPLE0` directory. But if you want to build your own controller, you should start up with example 1 which is the default controller of the package (also available in `SIM/EXAMPLES/EXAMPLE1` directory). This example shows how to read the inputs and how to write to the outputs of the robot. It implements a very simple control algorithm. It will be more detailed in the following tutorial section.

3.3.3 User info setup

As a user programmer of *Khepera Simulator*, you will need an area to display some of your variables, results, graphs, explanations, etc. This area exists and allow you to write numerical values, text, drawings, etc. into *Khepera Simulator* main window. All the information you can write is divided into directories which contain pages. A default directory of three pages contains a description of *Khepera Simulator*. You can switch between directory by pressing the button "`info`". You can turn the pages by pressing the buttons "`+`" or "`-`" next to the "`info`" button. You can create up to 4 user directories, each one containing up to 255 pages ! In order to define this, you must edit the file `user_info.h` which is in `USER` directory. After that, you will have to fill in the function `DrawUserInfo(struct Robot *robot,char info,char page)` where `info` is the info directory and `page` is the current page of this directory. Here are the constants to be edited in `user_info.h` header file:

- `NUMBER_OF_INFO` is the number of user information directories your want. This value must be between 0 and 4.
- `PAGES_INFO_x` is the number of pages for the directory number `x` (`x` ranging from 1 to 4). These values must be between 0 and 255.
- `TITLE_INFO_x` is the title of the directory `x` (`x` ranging from 1 to 4).

3.4 A tutorial example

3.4.1 Foreword

Four examples are given within *Khepera Simulator* package. The are located in the directory `SIM/EXAMPLES/EXAMPLEx` where `x` ranging from 0 to 3 is the number of the example. Example 0 is not really an example of a controller since it contains all the necessary functions for *Khepera Simulator* to run, but these functions are empty, resulting in an "empty" controller. Only example 1 will be explained in this tutorial. For all the examples, read the `readme` files that are in the directories `EXAMPLES/EXAMPLEx`. They contain indications about the installation and the compilation of the examples. Here is a short description of the examples:

- *Example 0*: the "empty" controller example.
- *Example 1*: an example of simple control algorithm.
- *Example 2*: artificial neural networks and gnuplot. This example shows how to implement artificial neural networks to drive the robot. It also features a pipe to gnuplot utility in order to display graphs (here the path of the robot). The neural networks shown here are resulting from an evolutionary process using genetic algorithms, morphogenesis, and artificial metabolism described in [2].
- *Example 3*: *Khepera Simulator* multi-agents module developed by Manuel Clergue allows to control several simulated *Khepera* robots.
- *Example 4*: *Khepera Simulator* simulated serial device module. This module is especially useful if you already developed a program sending serial commands to a real Khepera through the serial link of your computer. You will just need to redirect the input and output serial streams to *Khepera Simulator* pipes files and you will be able to observe the simulated *Khepera* driven by your serial commands. A list of the commands supported by *Khepera Simulator* is available in appendix C.

3.4.2 Let's program a *Khepera* robot controller !

This tutorial shows the implementation of a very simple control algorithm inspired from Braitenberg [1]. The source files are in `EXAMPLES/EXAMPLE1/USER/` directory.

First of all, let's define our algorithm:

```
repeat
  o If the robot perceive no obstacle, then move forwards.
  o If an obstacle is perceived on the left hand side of the robot, then turn
    to the right.
  o If an obstacle is perceived on the right hand side of the robot, then turn
    to the left.
until something is detected in the back of the robot.
```

To program this into *Khepera Simulator*, we need to translate "perceive no obstacle" into something dealing with the sensors of the robots. The sensors of the robot are readable through the variable `robot` (type `struct Robot`). The value corresponding to the distance measurement of the front sensor 2 (see figure 3) is stored in: `robot->IRSensor[2].DistanceValue` (type int). Values range between 0 and 1023. So if this value exceeds a given threshold, say 900 (which will be defined in `COLLISION_TH` constant), one can consider that an obstacle has been detected by this sensor.

In order to drive the motors of the robot, one must write in the variable `robot` the values corresponding to the speed of each motor we want to apply. These two integers will be written in `robot->Motor[LEFT].Value` and `robot->Motor[RIGHT].Value`. They range between -10 and $+10$. We will define as constants in this range a `TURN_SPEED` and a `FORWARD_SPEED`.

All the input and output operations must occur within the boolean function `StepRobot(struct Robot *robot)`. This function returns `FALSE` to stop the run of the robot and `TRUE` otherwise:

```
#define FORWARD_SPEED   5                    /* normal (slow) forward speed */
#define TURN_SPEED      4                    /* normal (slow) turn speed    */
#define COLLISION_TH    900                  /* value of IR sensors to be   */
                                             /*   considered as collision   */

boolean StepRobot(struct Robot *robot)
{
  if ((robot->IRSensor[0].DistanceValue > COLLISION_TH) ||   /* front left */
      (robot->IRSensor[1].DistanceValue > COLLISION_TH) ||   /*  sensors   */
      (robot->IRSensor[2].DistanceValue > COLLISION_TH))
                                              /* if there is a collision on the
                                                   left side of the robot */
  {
    robot->Motor[LEFT].Value  =  TURN_SPEED;
    robot->Motor[RIGHT].Value = -TURN_SPEED;          /* turn right */
  }
  else if ((robot->IRSensor[3].DistanceValue > COLLISION_TH) ||
           (robot->IRSensor[4].DistanceValue > COLLISION_TH) ||
           (robot->IRSensor[5].DistanceValue > COLLISION_TH))
                                              /* if there is a collision on the
                                                   right side of the robot */
  {
    robot->Motor[LEFT].Value  = -TURN_SPEED;
    robot->Motor[RIGHT].Value =  TURN_SPEED;          /* turn left */
  }
  else
```

```
  {
    robot->Motor[LEFT].Value  = FORWARD_SPEED;
    robot->Motor[RIGHT].Value = FORWARD_SPEED;  /* else go forward (default) */
  }

  if ((robot->IRSensor[6].DistanceValue > COLLISION_TH)||  /* collision in */
      (robot->IRSensor[7].DistanceValue > COLLISION_TH))   /*   the back   */
   return(FALSE);  /* stop */
  else
   return(TRUE);   /* continue */
}
```

With this simple code, we have defined a complete robot controller. It will run in a loop when pressing the "`run`" button or run once when pressing the "`step`" button. You can now compile example 1 on your computer. Normally, it is the default example installed within the package, so you just need to type `make`. This will create the object files in `SIM/OBJ/` directory and produce the executable file: `sim`. Type `sim` to run *Khepera Simulator*.

In order to display some text, numerical values, or drawings in the user info area, have a look at the function `DrawUserInfo(...)` in the source file `user.c` which is in `SIM/EXAMPLE1/USER/` directory. It seems so simple that I wouldn't describe it here. A lot of graphical functions are available to let your imagination as free as possible (see appendix A.3 for more information).

3.5 Multi Agent Package

3.5.1 Overview

The purpose of this extension of the *Khepera Simulator* is to operate a group of robots instead of a single robot. This group of robots is viewed by the simulator like a single entity, a "C" structure (`Multirobots`). Each robots of the entity react according to a user specified behavior. All robots may have the same behavior or some robots may have a specific behavior, as you need. The actions of the robots at a defined time are calculated in a synchonous way, that is robots move one after another.
This package works only with simulated robots.

3.5.2 Implementation

The "C" code concerning Multi Agent can be found in the files multirobots.c and multirobots.h, placed in the directory CONTRIB. These files define the structure `Multirobots` and some useful functions. You may see the file multirobots.h for description of these functions (see appendix D for a paper version).
The structure is composed of three fields. A field for storing the number of robots of the group. This number is fixed at the creation of the structure and should not change until the structure is destroyed. Another field is an array of pointers on robots. These pointers, given by a malloc operation, should not change. On the contrary, structures pointed by the pointers may change in the way you want, if they remain `Robot` structures. The third field is a marker which indicate the number of the current robot.
The most important functions are `MultiRobotRun` and `MultiRobotRunFast`, which execute a cycle of robots move. When you call these functions with a structure `Multirobots`, they call `StepMultiRobots` and `FastStepMultiRobots` for each robots, each time changing the number of the current robot.
The functions `StepMultiRobots` and `FastStepMultiRobots` receive a structure `MultiRobots` as a parameter. They should apply a `StepRobot`-like function on the current robot. To have robots with different behaviors, just write several `StepRobot`-like functions and call them according to the current robot.

Two versions are provided to allow a run with graphical display (`MultiRobotRun`) and a faster run without such display (`MultiRobotRunFast`).

3.5.3 How to use it?

In the same way you do for the simple simulator, you have to write a file user.c, in which you define some functions. In addition of the general ones (i.e. those of the simple simulator), you have to implement two specific functions:

```
boolean FastStepMultiRobots(struct MultiRobots *multi);
boolean StepMultiRobots(struct MultiRobots *multi);
```

To use a structure `MultiRobots`, you have to create it using the function `CreateMultiRobots`. This function has the number of robots you want in the group as a parameter.
Then, you may run the robots using the functions `MultiRobotRun` and `MultiRobotRunFast`. These functions, which are defined in multirobots.c, make use of the functions `StepMultiRobots` and `FastStepMultiRobots` to determinate the behavior of the current robot of the group.
After use, you have to call `FreeMultiRobots` in order to free memory.
You are greatly encouraged to have a look at the example 3, before coding your own multi agent simulation.

3.5.4 future works

The major lack of this package is the impossibility to command several real robots. Further versions of the simulator should allow this feature.

3.6 Author's Notes

I am are aware that the models for the sensors and for the motors are very simple. I choose computer efficiency instead of precision, making this simulator suitable for computer expensive algorithms, especially genetic algorithms.

I do not handle real time problems in the tutorial example presented here because the controller is very simple and doesn't need any synchronization. Anyway, it is interesting to know that at a speed of 10 (on both motors), the simulated robot covers exactly 5 millimeters for one simulation step, while the real robot cover an unknown distance (depending on many factors including computer speed, the control algorithm complexity, the serial link, etc.). This may give ideas to build a system taking care of real time problems.

4 Acknowledgments

I developed this software during my Ph-D at i3S laboratory with professor Joëlle Biondi and assistant professor Philippe Collard as Ph-D directors (Mage Team). Manuel Clergue, a Ph-D student studying genetic algorithms and evolutionary neural networks in our laboratory, was the first user and beta-tester of this software. He also developed the multi-agents module included in example 3. I am grateful to all these people for their assistance and some precious advices during the development of the software. Moreover, I would like to congratulate the designers of the *Khepera* robot: Edo Franzi, André Guignard and Francesco Mondada (K-Team SA, Preverenges, CH) for their brilliant realization.

References

[1] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, 1984.

[2] Olivier Michel and Joëlle Biondi. Morphogenesis of neural networks. *Neural Processing Letters*, 2(1), January 1995.

[3] F. Mondada, E. Franzi, and P. Ienne. Mobile robot miniaturisation: A tool for investigation in control algorithms. In *Third International Symposium on Experimental Robotics*, Kyoto, Japan, October 1993.

Appendix

A Library of functions

A.1 Data structures

You need to know three C structures to drive *Khepera* robot which are defined in `robot.h` file in `SIM/SRC/` directory:

```
struct Motor
{
  double    X,Y,Alpha;
  short int Value;         /* motor speed between -10 and +10 */
};

struct IRSensor
{
  double    X,Y,Alpha;
  short int DistanceValue;  /* typically between 0 and 1023 */
  short int LightValue;     /* typically between 0 and 500  */
};

struct Robot
{
  char                  Name[16];
  double                X,Y,Alpha; /* X and Y (millimeter), Alpha (rad) */
  double                Diameter;
  u_char                State;
  struct Motor          Motor[2];  /* use RIGHT & LEFT instead of 0 & 1 */
  struct IRSensor       IRSensor[8]; /* see simulated robot on figure 1 */
};
```

A.2 Fill-in functions

These functions needs to be fulfilled in order to attach actions to the buttons of the graphical interface.

```
void NewRobot(struct Robot *robot)
  This function is called when the "NEW ROBOT" buttons is pressed.

void LoadRobot(struct Robot *robot,FILE *file)
  This function allows the user to write some data in the robot file, using
  the C functions fprintf(file,pattern,data). It is called when the "LOAD
  ROBOT" button is pressed.

void SaveRobot(struct Robot *robot,FILE *file)
  This function allows to retrieve the data saved by the LoadRobot function
  using the C function fscanf(file,pattern,data). It is called when the "SAVE
  ROBOT" button is pressed. Both functions need to be updated in the same way
  (they must load and save exactly the same datas in the same order).

void RunRobotStart(struct Robot *robot)
```

```
  This function is called once when the "RUN ROBOT" button is pressed.

boolean StepRobot(struct Robot *robot)
  This function is called as long as the "RUN ROBOT" button is down.

void FastStepRobot(struct Robot *robot)
  This function is called by FastRunRobot().

void RunRobotStop(struct Robot *robot)
  This function is called when the "RUN ROBOT" button is released.

void ResetRobot(struct Robot *robot)
  This function is call when the "RESET ROBOT" button is pressed.

void UserCommand(struct Robot *robot,char *text)
  This function is called when the "COMMAND" button is pressed. The text
  parameter passed to it is the string that the user typed on the keyboard.
  It is a powerful way to do anything you want (set parameters, start various
  algorithms, etc.).

void DrawUserInfo(struct Robot *robot,char info,char page)
  This function is called each time the program needs to redraw the display
  of the window. It must contain all the drawings and texts for the user info
  box.

void UserInit(struct Robot *robot)
  This function is called at the beginning of the program. It can be used to
  make some initializations.

void UserClose(struct Robot *robot)
  This function is called at the end of the program. It allows to close some
  file eventually open during UserInit() or to free some memory.
```

A.3 Graphical functions

These functions allow to draw text and graphics in the user info box. The coordinates (0,0) indicates the upper left corner of the box and the coordinates (500,400) indicates the lower right corner of the box.

```
void Color(char color)
  Sets the color of the pen. Available colors are:
  BLACK, DIM_GREY, GREY_69, GREY, LIGHT_GREY, WHITE, BLUE, BLUE_CYAN, CYAN,
  CYAN_GREEN, GREEN, GREEN_YELLOW, YELLOW, YELLOW_RED, RED, MAGENTA,
  LIME_GREEN, BROWN, MAROON, GOLD, AQUAMARINE, FIREBRICK, GOLDENROD,
  BLUE_VIOLET, CADET_BLUE, CORAL, CORNFLOWER_BLUE, DARK_GREEN,
  DARK_OLIVE_GREEN, PEACH_PUFF, PAPAYA_WHIP, BISQUE, AZURE, LAVENDER,
  MISTY_ROSE, MEDIUM_BLUE, NAVY_BLUE, PALE_TURQUOISE and SEA_GREEN.

void FillRectangle(int x,int y,int width,int height)
  Draws a filled rectangle with the upper left corner at (x,y).

void DrawLine(int x1, int y1, int x2, int y2)
  Draws a line between (x1,y1) and (x2,y2).
```

```
void DrawPoint(int x,int y)
  Draw a point at (x,y).

void DrawRectangle(int x,int y,int width,int height)
  Draws an empty rectangle with the upper left corner at (x,y).

void FillArc(x,y,width,height,angle1,angle2)
  Draws a filled arc with the upper left corner at (x,y) between
  (angle1 / 64) and (angle2 / 64) in degrees.

void DrawArc(x,y,width,height,angle1,angle2)
  Draws an empty arc with the upper left corner at (x,y) between
  (angle1 / 64) and (angle2 / 64) in degrees.

void DrawText(int x,int y,char *text);
  Draw text at (x,y)

void UndrawText(int x, int y,char *text);
  Undraw text at (x,y)

void WriteComment(char *text)
  Write a comment at the comment line.

void EraseComment()
  Erase the comment on the comment line.

void DrawRobot(struct Robot *robot)
  Redraw the simulated robot in its environment.

void ShowUserInfo(int info,int page);
  Display a page of the user info box.

u_char GetUserInfo()
  Returns the current user info number (ranging from 1 to 4).

u_char GetUserInfoPage()
  Returns the current page number of user info (ranging from 0 to 255).
```

A.4 Other useful functions

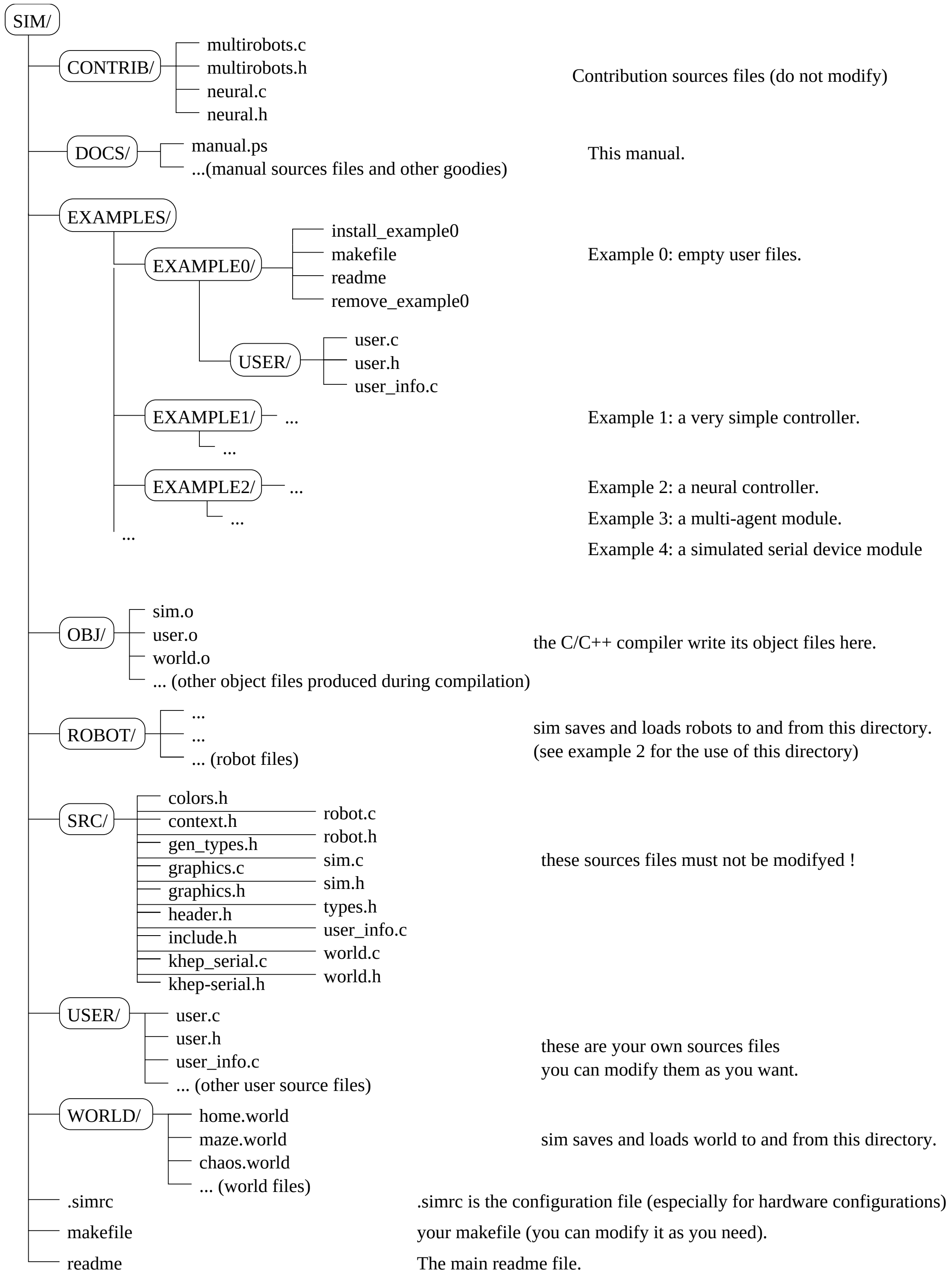
These functions should be called by the UserCommand() function or sub-functions.

```
boolean StopCommand()
  returns TRUE is the "COMMAND" button is released.

void FastRunRobot(struct Robot *robot)
  runs the simulated robot without displaying it (faster).

boolean RunRobot(struct Robot *robot)
  Runs the simulated robot. Returns FALSE if the robot is stopped.
```

B Directory structure



C Khepera Serial Commands

```
A  Configure (not implemented in Khepera Simulator)

B  Read software version (not implemented in Khepera Simulator)

D  Set speed
Format of the command:  D, speed_motor_left, speed_motor_right<\n>
Format of the response: d<\n>
Effect:                 Set the speed of the two motors. The unit is the
                        pulse/10 ms that correspond to 8 millimeters per
                        second on the real robot.

E  Read speed
Format of the command:  E</n>
Format of the response: e, speed_motor_left, speed_motor_right<\n>
Effect:                 Read the instantaneous speed of the two motors. The
                        unit is the pulse/10 ms that correspond to 8
                        millimeters per second on the real robot.

G  Set position (not implemented in Khepera Simulator)

H  Read position (not implemented in Khepera Simulator)

N  Read proximity sensors
Format of the command:  N</n>
Format of the response: n, val_sens_left_90,val_sens_left_45,val_sens_left_10,
                        val_sens_right_10,val_sens_right_45,val_sens_right_90,
                        val_sens_back_right,val_sens_back_left<\n>
Effect:                 Read the 10 bit values of the 8 proximity sensors, from
                        the front sensor situated at the left of the robot,
                        turning clockwise to the back-left sensor.

O  Read ambient light sensors
Format of the command:  O</n>
Format of the response: o, val_sens_left_90,val_sens_left_45,val_sens_left_10,
                        val_sens_right_10,val_sens_right_45,val_sens_right_90,
                        val_sens_back_right,val_sens_back_left<\n>
Effect:                 Read the 10 bit values of the 8 ambient light sensors,
                        from the front sensor situated at the left of the
                        robot, turning clockwise to the back-left sensor.
```

D multirobots.h

```
/****************************************************************************/
/* File:        multirobots.h (Khepera Simulator)                          */
/* Author:      Manuel CLERGUE <clerguem@alto.unice.fr>                    */
/* Date:        Thu Jan 16 14:39:05 1996                                   */
/* Description: Extension of Khepera Simulator                             */
/*              for multi-agents simulation                                */
/* Copyright (c) 1995                                                      */
/* Olivier MICHEL                                                          */
/* MAGE team, i3S laboratory,                                              */
/* CNRS, University of Nice - Sophia Antipolis, FRANCE                     */
/*                                                                         */
/* Copyright (c) 1996                                                      */
/* Manuel CLERGUE                                                          */
/* MAGE team, i3S laboratory,                                              */
/* CNRS, University of Nice - Sophia Antipolis, FRANCE                     */
/*                                                                         */
/* Permission is hereby granted to copy this package for free distribution. */
/* The author's name and this copyright notice must be included in any copy. */
/* Commercial use is forbidden.                                            */
/****************************************************************************/

#ifndef MULTIROBOTS_H
#define MULTIROBOTS_H

#define D_MAX         50.0   /* max. dist. between 2 robots for mutual perc. */
#define IR_MAX        1023   /* max. value of IR captors */

/* useful external functions */
extern void    DrawLittleRobot(struct Robot *sr,struct Robot *r);
extern void    ChooseRandomPosition(struct World *world,double *x,double *y,
                                    double *alpha);
extern u_short IRSensorDistanceValue(struct World *world,short int x,short int y,
                                     double alpha);
extern u_short IRSensorLightValue(struct World *world,short int x,short int y,
                                  double alpha);


/* The MultiRobots structure manage the use of several robots */
/* Some services are provided with this structure */
/* It is highly recommended to use them (or to create others) */
/* instead of using those of the Robot structure */

struct MultiRobots
{
  struct Robot **robots;  /* Array of pointers to Robot */
  short int     current;  /* Current Robot being treated -- use this carefully */
  short int     number;   /* Number of Robots in the structure */
};

/* CreateMultiRobots create and initialyze the structure with number Robots */
MultiRobots *CreateMultiRobots(long int number);

/* Free the structure and the Robots */
void FreeMultiRobots( MultiRobots *multirobots);

/* Calculate the influence of the group on the current Robot's captor (xc,yc,alpha) */
```

```
/* Used in MultiInitSensors */
short int MutualInfluence(short int xc,short int yc,double alpha,
                            short int value,MultiRobots *multirobots);

/* Calculate the value of IR-Distance captors of the Robots in the structure */
/* Used in MultiRobotRunFast */
void MultiInitSensors(Context *context,MultiRobots *multirobots);

/* This is the most useful service. This is THE one you have to use */
/*  in standart situations */
/* There is two version (Fast and Normal) */
/* The fast one (without the display of robots on the screen) use
/*     FastStepMultiRobots(MultiRobots *multirobots)*/
/* The other one use StepMultiRobots(MultiRobots *multirobots) */
/* You have to implement these functions in the same way that you have to do it
/* for one robot */
void MultiRobotRunFast(Context *context,MultiRobots *multirobots);
void MultiRobotRun(Context *context,MultiRobots *multirobots);

/* nothing important */
double DistanceBetRobots(Robot *rob1,Robot rob2);

/* This service allow you to place the robots at random place in the world */
void PlaceRobots(Context *context,MultiRobots *multirobots);

/* The functions you have to implement */
extern boolean FastStepMultiRobots(struct MultiRobots *multi);
extern boolean StepMultiRobots(struct MultiRobots *multi);
#endif
```