Manual Locokit over-a-network controller:

The project is made to control Locokit robot by using an interface that connects with the robot over a wireless connection. The project consists of two programs: the server program and the client program. The server program is executed on the robot's processor. The client program provides an interface that enables the user to send commands to the server to be executed. The used protocol for the connection is TCP to make sure that all transmitted information are preserved.

Running the system and establishing the connection:

The following steps demonstrate how the system should be used:

1) connecting to a wireless network:
The robot connects automatically with the wireless network “gumstix-network” and it has the IP 192.168.2.4 (the IP could be different for a different locokit robot instance). Connect the client computer to the same network and give your pc different IP address.
(for more details: http://modular.tek.sdu.dk/index.php?page=connecting-to-locokit-using-wifi)

2) Establish an SSH session:
open a terminal and write:
ssh root@192.168.2.4 (the IP could be different for a different locokit robot instance)
to open an ssh session. No password is needed for the login so just press enter when you are asked about it.

3) Run the server project:
open the following directory:
$: users/Bassel/wlan_controller
inside this folder there are the following files:

- Makefile //the configuration of the compiling process
- actuateMotors.c //the server code
- loco-settings.cfg //the robot configuration file

the code can be compiled by entering the folder and enter: make all
the compiler generate a file called “actuateMotors“
(if you are using a new instance of the robot, add those files to any directory and compile them and continue. The files can be found under utils/real_robots/locokit)
- run the server program by typing “./actuateMotors”. The server at that point will just wait for a connection from the client (socket). you should see the sentence “waiting for connections” on the terminal after running the executable file.

4) Run the Client program:

To connect the client program just open ANOTHER terminal and go to your client program executable file and run it.
NOTE: the server program should be already executed before the client program starts.

5) sending commands:
use your “LocoKitInterface” class to send and receive commands between the client and the server. The available commands will be showed in the next section.

6) Closing:
make sure that the constructor of your client LocoKitInterface object is called before you terminate your client program (i.e. use delete if you define a pointer to the object LocoKitInterface). After the client program is terminated the server program can be terminated also.

How to use the interface:

Two classes (LocoKitInterface and ConnectionClass) are provided to the user. These two classes enable the client to send commands to the server. The user should instantiate the class “LocoKitInterface” which uses the class ConnectionClass as a lower layer to send commands.

Steps:
1) define an LocoKitInterface object:

```
LocoKitInterface LKI; //define an object
if (LKI.establish_connection() == -1) {
        printf("Error from LocoKitInterface: a connection couldn't be
                                                established...\n");
        return -1;
}
```

2) use the available functions:
e.g. setActuatorPWM

```
LKI.setActuatorPWM(pmw, 2);
```

Available functions (Client side):

```
int establish_connection();
//Establish the connection with the server side.
//Return 0 if successful, -1 for errors.
```

```
int setActuatorPWM(float pwm, int actuator);
//Set the PWM of an actuator.
//Parameters:
//pwm: The PWN in the range from -1024 to 1024
//Actuator: The ID of the actuator
//Return 0 if successful, -1 for errors.

int setActuatorStopped(int actuator);
//Stop an actuator.
//Parameters:
//actuator: The ID of the actuator
//Returns:    0 if successful, -1 for errors

int setConstantSpeedInterpolatingFunction (int actuator, float period, float
phaseOffset, int directionNegative);
//Set the motor control interpolation values.
//Parameters:
//actuator:   The motor to control
//period: The period time in seconds
//phaseOffset: The phaseOffset in degrees
//directionNegative: The direction of rotation, 0 for rotation in positive direction 1
for negative
//Returns: 0 if successful, -1 for errors


int getNumberOfSensors();
//returns the number of sensors.
//Returns 0 if successful, -1 for errors

int getSensorValueRawFloat(int sensor, float &sensor_value);
//get the current unmodified value of the sensor (returned in the "sensor_value"
parameter)
//Parameters: The ID of the sensor
//returns: 0 if successful, -1 for errors
//sensors ID numbers:
//sensor (0) ---> acc on x
//sensor (1) ---> acc on y
//sensor (2) ---> acc on z
//sensor (3) ---> gyo x
//sensor (0) ---> gyo y
```

```
//sensor (0) ---> gyo z


int updateSensorValueRawFloat_array();
//provides 6 values that are pointed out in the previous
//function (getSensorValueRawFloat ID:0..5) in one array and at the same order.
//these values are saved in the public member sensory_inputs.
//returns 0 if successful, -1 otherwise

int     getActuatorVelocity(int        actuator, float&     velocity);
//Get the velocity of an actuator.
//Parameters:
//actuator: the ID of the actuator
//velocity: the       velocity in degrees per second in        the parameter “velocity”
//returns 0 if successful, -1 otherwise

int getActuatorPosition(int actuator, float& position);
//Get the current position of an actuator.
//Parameters:
//actuator: The ID of the actuator
//provide the position in degrees in the parameter “position”
//returns 0 if successful, -1 otherwise

int getActuatorPWM(int actuator, float& PWM_value);
//get the PWM of an actuator
//Parameters:
//actuator: the ID of the actuator
//provides the PWM in the parameter "PMW_value"
//returns 0 if successful, -1 otherwise

int getSensorValue(int sensor, float &sensor_value);
//Get current sensor value.
//Parameters:
//sensor: the ID of the sensor
//provides the current value of the sensor in the parameter "sensor_value"
//returns 0 if successful, -1 otherwise

int resetSensorValue(int sensor);
////Reset a sensor. The current sensor value is used as offset for future calls to
getSensorValue but not getSensorValueRaw.
//Parameters:
```

//sensor: the ID of the sensor
//returns 0 if successful, -1 otherwise

int terminate_connection_with_server();
//This function is called to terminate the connection with the server. It send a command to the //server to terminate the server program also

Available Functions (Server side):

void PWM_control_routine(**int** act_number, **float** pwm);
//applying the pwm value on the targeted actuator.
It takes the actuator ID as an input in addition to the pwm value

void ConstantSpeedInterpolatingFunction_control_routine(**int** act_number, **float** period, **float** phaseoffset, **int** direction);
//applying the ConstantSpeedInterpolatingFunction values on the targeted actuator
//it sets the motor control interpolation values.
//Parameters:
//actuator: The motor to control
//period: The period time in seconds
//phaseOffset: The phaseOffset in degrees
//directionNegative: the direction of rotation, 0 for rotation in positive direction 1 for negative

void setActuatorStopped_routine(**int** act_number);
//stop a targeted actuator. Takes the actuator number as an input

int setActuatorPWM_command(**int*** socket);
//receives setActuatorPWM parameters from the client and calls the required functions //to execute the command.
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int setConstantSpeedInterpolatingFunction_command(**int*** socket);
//receives setConstantSpeedInterpolatingFunction parameters from the client and calls the required functions to execute the command (**ConstantSpeedInterpolatingFunction_control_routine**).
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

```
int setActuatorStopped_command(int* socket);
//receives the targeted motor ID from the client and calls the required functions to
execute the command (stop command for a specific motor).
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getNumberOfSensors_procedure(int* socket);
//calls the required functions to get the number of sensors
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getSensorValueRawFloat_procedure(int* socket);
//receives the targeted sensor ID from the client and
//calls the required functions to execute the function
//"getSensorValueRawFloat". It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getSensorValue_procedure(int* socket);
//receives the targeted sensor ID from the client and
//executes the function "getSensorValue" which get a sensor value.
//It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int resetSensorValue_procedure(int* socket);
//receives the targeted sensor ID from the client and
//executes the function "resetSensorValue" which reset a sensor.
//It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getSensorValueRawFloat_array_procedure(int* socket);
//executes the function "getSensorValueRawFloat" for all available sensors
//to get all sensors' values.
//It sends back the results to the client as an array of 6 elements
//sensor (0) ---> acc on x
//sensor (1) ---> acc on y
//sensor (2) ---> acc on z
//sensor (3) ---> gyo x
//sensor (4) ---> gyo y
```

```
//sensor (5) ---> gyo z
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getActuatorVelocity_procedure(int* socket);
//receives the targeted motor ID from the client and
//executes the function "getActuatorVelocity" which gives the motor velocity.
//It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed


int getActuatorPosition_procedure(int* socket);
//receives the targeted motor ID from the client and
//executes the function "getActuatorPosition" which gives the motor position.
//It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int getActuatorPWM_procedure(int* socket);
//receives the targeted motor ID from the client and
//executes the function "getActuatorPWM" which gives the motor PMW.
//It sends back the results to the client
//It takes the connected socket numebr as an input
//return 1 if successful, -1 errors, 0 connection is closed

int command_interpreter(int command, int* socket);
//this function represent the protocol interpreter which
//interprete the commands that are coming from client.
//It takes the command number as an input and the client socket
//2 ----> PWM control
//3 ----> ConstantSpeedInterpolatingFunction control
//4 ----> get Number Of Sensors
//5 ----> get Sensor Value RawFloat
//6 ----> get Sensor Value RawFloat_array (returns an array)
//7 ----> get Actuator Velocity
//8 ----> get Actuator Position
//9 ----> get Actuator PWM
//10 ----> stop an Actuator
//11 ----> get Sensor Value
//12 ----> reset Sensor Value
//50 ----> termination code
```

General Notes:

1) if you are using a new instance of the robot don't forget to modify the micro IP_Address "192.168.2.4"
to the new address in the client code.
also don't forget to modify the loco-settings.cfg file to the new configurations of your new instance.

2) These programs use some system built-in functionalities (to open the socket, to use the connection buffers, etc). Therefore, some errors may occur. In case, you face error when connecting to the server, try please to restart your robot and your client computer and try again.

3) log file: "loco-settings.cfg" should be located on the same file in which the server executable file is located. It defines the robot's configurations which are used in the program. Any changes that you want to introduce into this file may affect the program. In addition, any hardware changes (e.g. new actuator) need to be added into this file.

4) AN example on how to use these programs is implemented for Locokit robot. The examples can be found under:
example 1: "examples/locokit/simple_control"
example 2: "examples/locokit/keyboard_control"

5) Adding more motors, use more than 4 motors

adding more index in loco-settings.cfg✉ ARM7MC = {0, 1 /*motor1*/, 2 /*motor2*/, 6 /*motor3*/, 12 /*motor4*/, xxx, xxx}

then you just pass the index number of the motor (the numbers that are written on the motors) that you want to send the control command to, as parameter in the function provided in the interface (e.g. setActuatorPWM).

For new robot

1) Added new network: network name (check from Locokit PC)
2) set IP address manually
3) Copy

Makefile //the configuration of the compiling process
actuateMotors.c //the server code

locosettings.cfg //the robot configuration file

In locosetting.cfg
change actuator number according to a Module number used!

# Actuators (used id 0 to enable broadcast):
ARM7MC = {0, 1, 2, 6, 12}

ARM7MC = {0, module number, module number, module number, module number}