Kilobots usage
Coding, compiling, deploying

Slides mostly from «Feri, Morganti, Morganti - Kilobots usage: coding, compiling and deploying»

HW and SW requirements

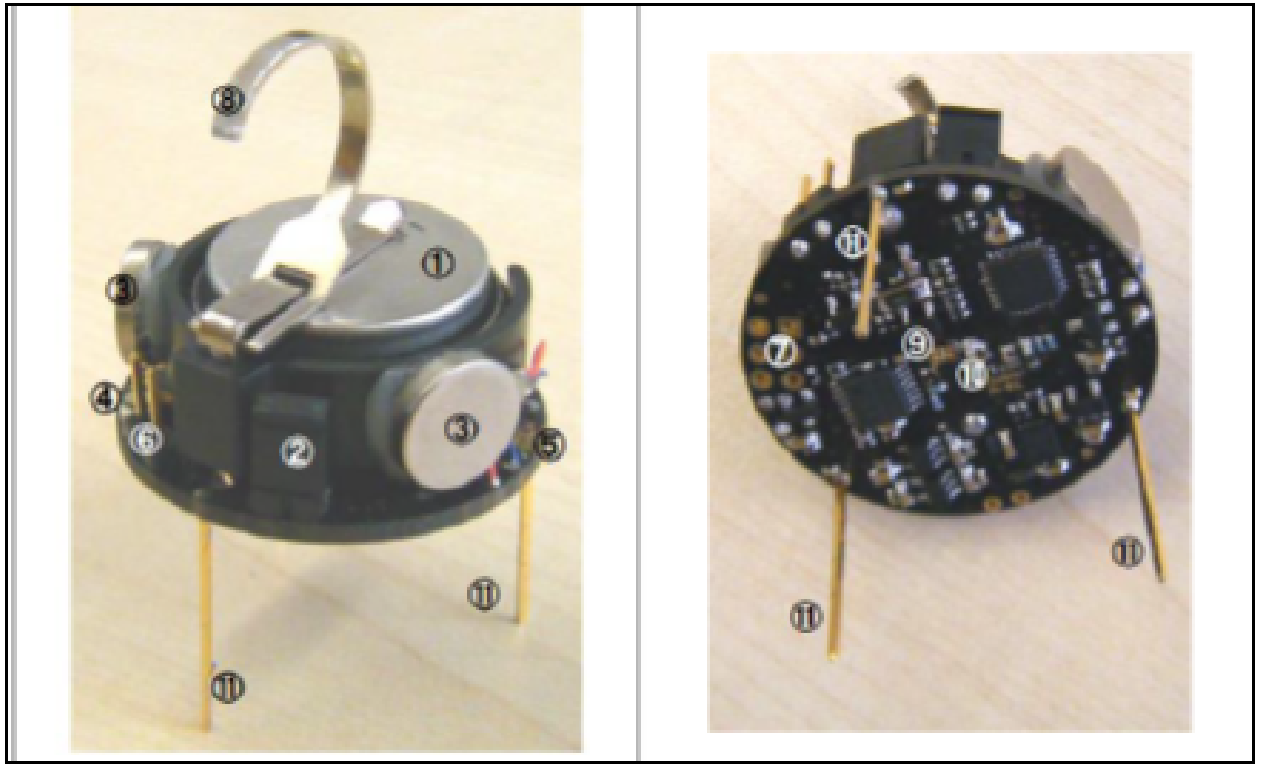
Hardware:

- Kilobots
- Over-head controller (OHC)
- Kilobotcharger
- (computer with an USB port/adapter)

Software: computer to be equipped with:

- We present installation instructions for Windows OS → but also Linux or OS can be used, see https://www.kilobotics.com/download , https://www.kilobotics.com/documentation
- Browser to access the online editor *https://www.kilobotics.com/editor*
- Dropbox Account (required to use the online editor)
- File libusb0.dll → **http://ftp.k-team.com/kilobot/CD**
- Zadig Software to modify the USB driver to work with new controller → **http://ftp.k-team.com/kilobot/CD** or https://zadig.akeo.ie/
- *kiloGUI*: Kilobot Controller program → *https://www.kilobotics.com*

Here are the kilobots!



① 3.7-Volt Battery (- up, + down)
② Power jumper
③ Vibration motors
④ LED (Red/Green/Blue)
⑤ Ambient light sensor
⑥ Serial output header
⑦ Direct programming socket
⑧ Charging Tab
⑨ IR Transmitter
⑩ IR Receiver
⑪ Robot leg

Specifications

Processor: ATmega 328p (**8bit @ 8MHz**)

Memory: **32 KB Flash , 1KB EEPROM**

Battery/ autonomy: Rechargeable Li-Ion 3.7V / 3-10 hours continuously, 3 months in sleep mode.

Communication: IR (up to 7cm, up to 32kb/s and 1kbyte/s with 25 robots), serial (256000 baud)

Sensing: 1 IR and 1 light intensity

Movement: forward, rotation (1cm/s , 45deg/s)

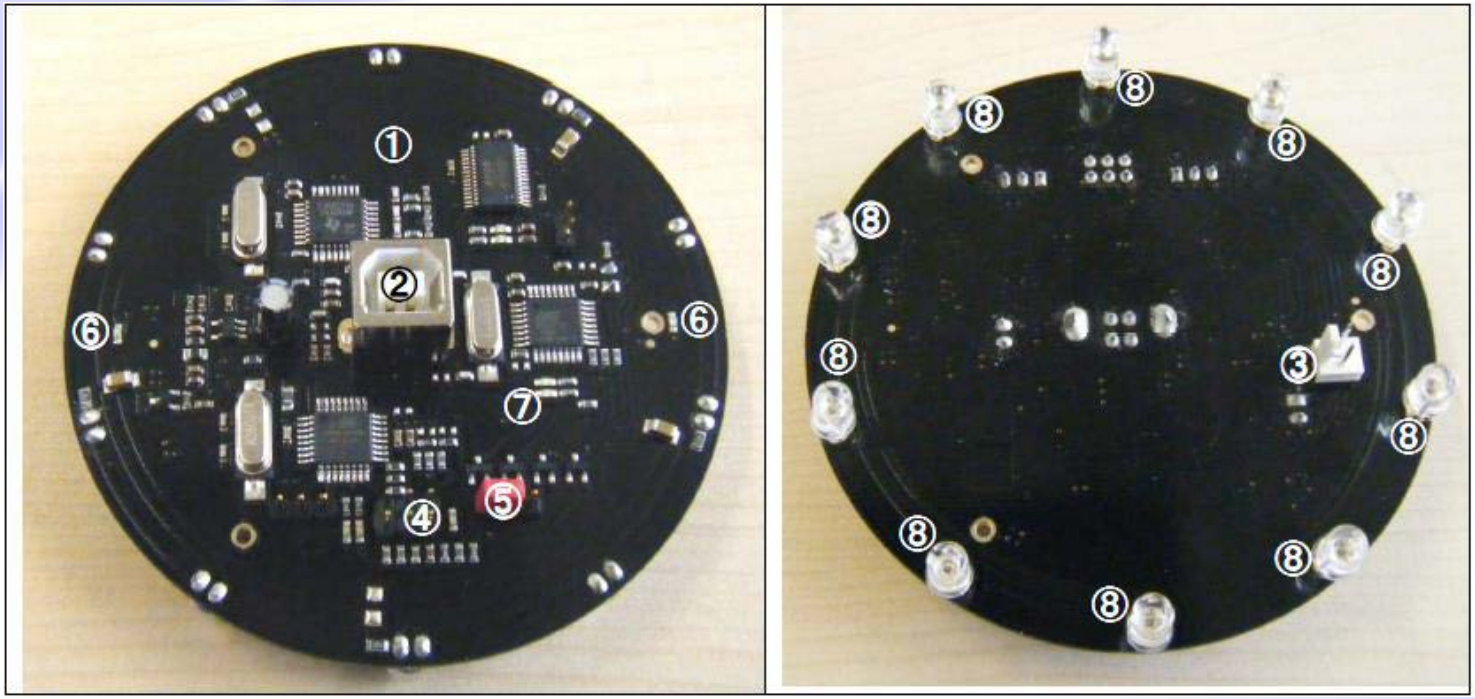
Light: one RGB led.

Software: Kilobot Controller software for controlling the robot

Programming: C language with WinAVR compiler combined with Eclipse or the online Kilobotics editor.

Dimensions: diameter: 33 mm, height 34 mm (including the legs, without recharge antenna).

The overhead controller (OHC)

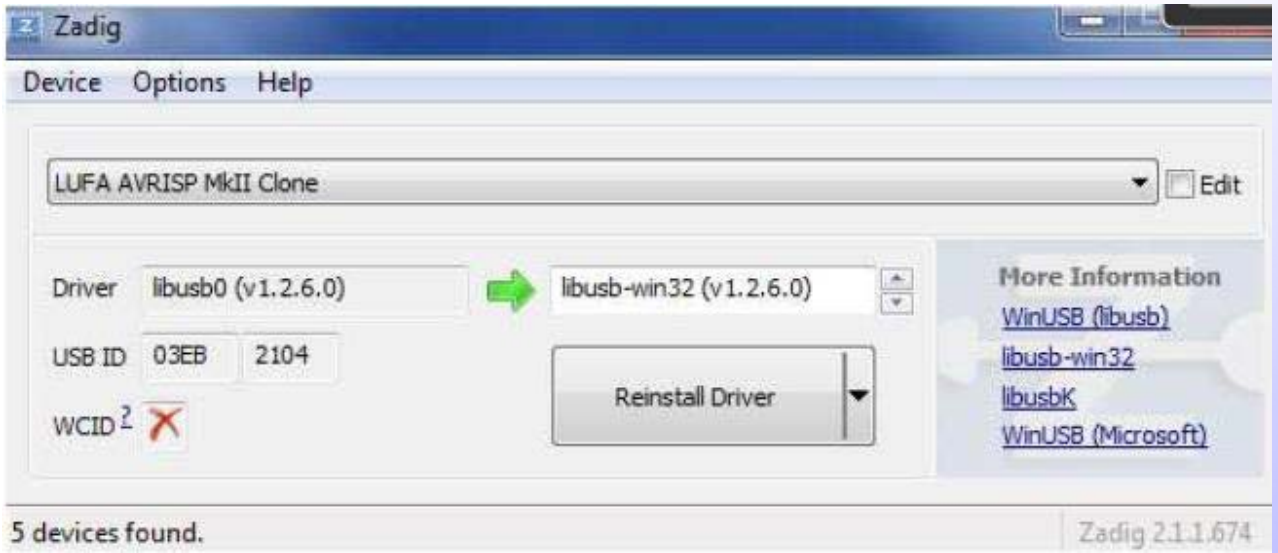


① Overheadcontroller board
② Connector for USB cable
③ Connector for debug cable
④ Connector for firmware programming
⑤ Firmware programming jumper
⑥ Diode for OHC connection test
⑦ Power-on LED
⑧ IR LED

OHC installation -1

If your OHC is not recognize by the KiloGUI interface, you will need to modify the driver:

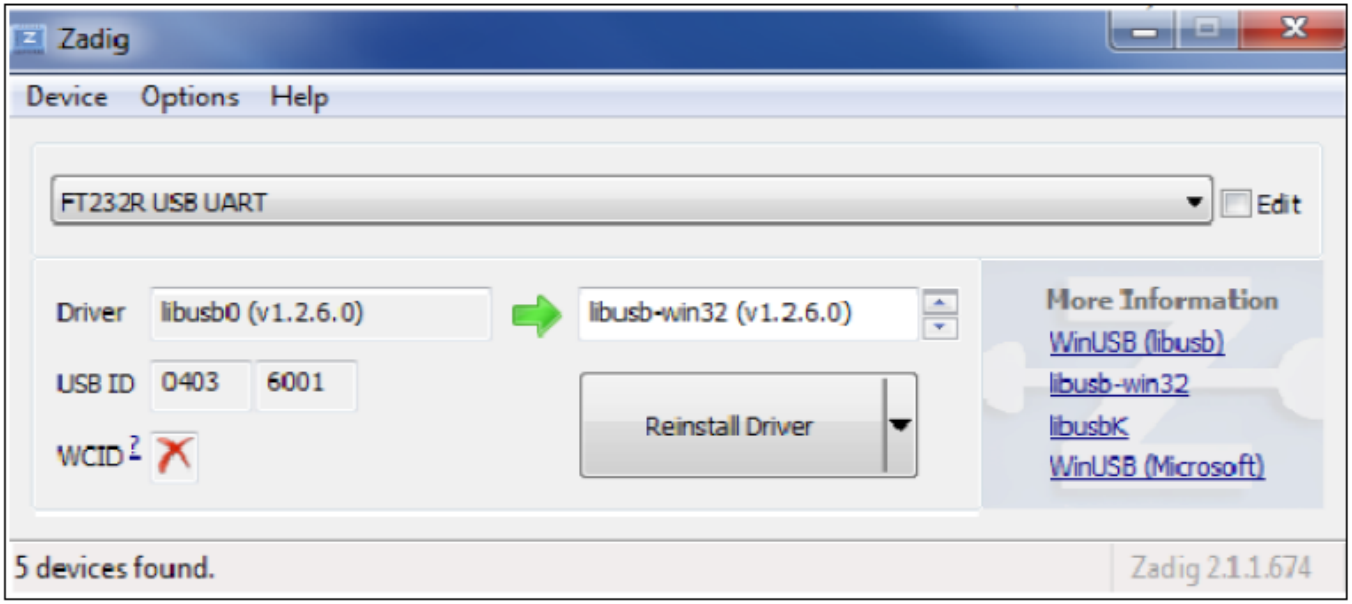
1) Executethe software «Zadig»
2) Option -> list all devices
3) Select « ***LUFA AVRISP MkIIClone*** » and « ***libusb-win32 (V1.2.6.0)***»



OHC installation

4) Click « **Replace Drive**» (or "**Reinstall Driver**" if already done once).

5) Select «**FT232R USB UART**» and «**libusb-win32 (V1.2.6.0)**»



6) Replace Driver

7) Close « Zadig» interface

Now the KiloGUI interface must (should?) recognize your OHC.

OHC installation verification - 1

1. Check that the firmware jumper on the OHC is plugged at the correct place (i.e., do not move the jumper from its usual place).

2. Connect the USB cable to the PC; the power LED of the OHC should turn green.

3. Run the "***kilogui.exe***" program, the message "***connected***" must appears at the bottom of the windows

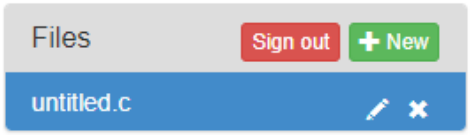
OHC installation verification - 2

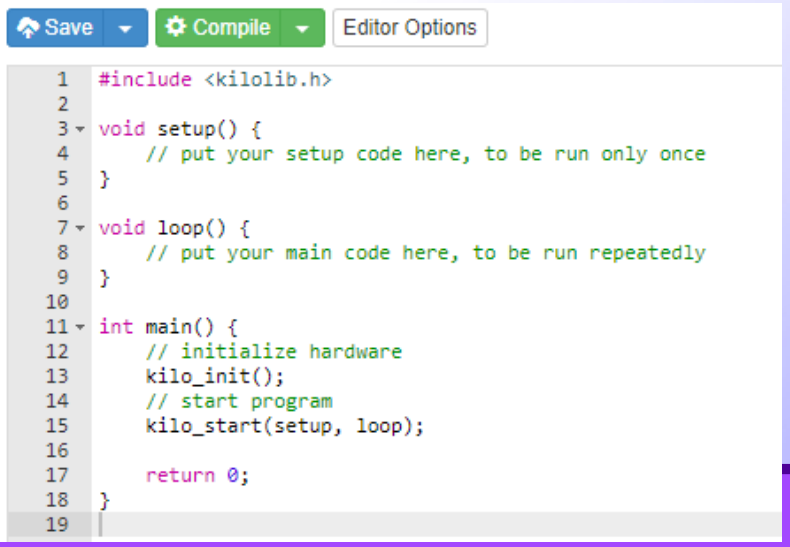
4. Press the ***"LedToggle"*** button to validate the communication (a Green LED on the OHC must change its state at each pressing).

5. If the message ***"connected"*** didn't appear, use Zadig to modify the driver.

USAGE: Write and build our Kilobot program (kilobotics.com)

1. Use the editor https://www.kilobotics.com/editor to create a new file
2. Rename this file to something you like, and select it for editing
3. You can compile the file, by clicking on the green Compile button. This will produce a `yourfilename.hexfile`
   - Use as example: https://github.com/SSR-Harvard/kilobotics-labs/blob/master/blink_led.c
4. Both code and compiled files are stored in Dropbox/Apps/KiloEdit
5. Now use the KiloGUI to upload the hex file to your kilobots

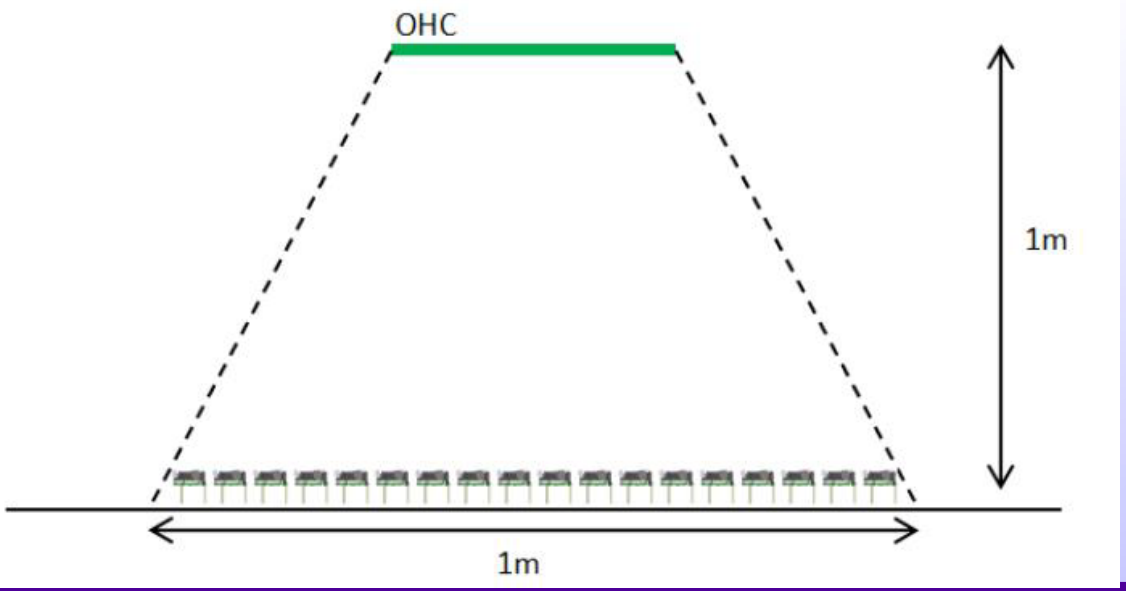




USAGE: Uploading and executing code in a group of Kilobots

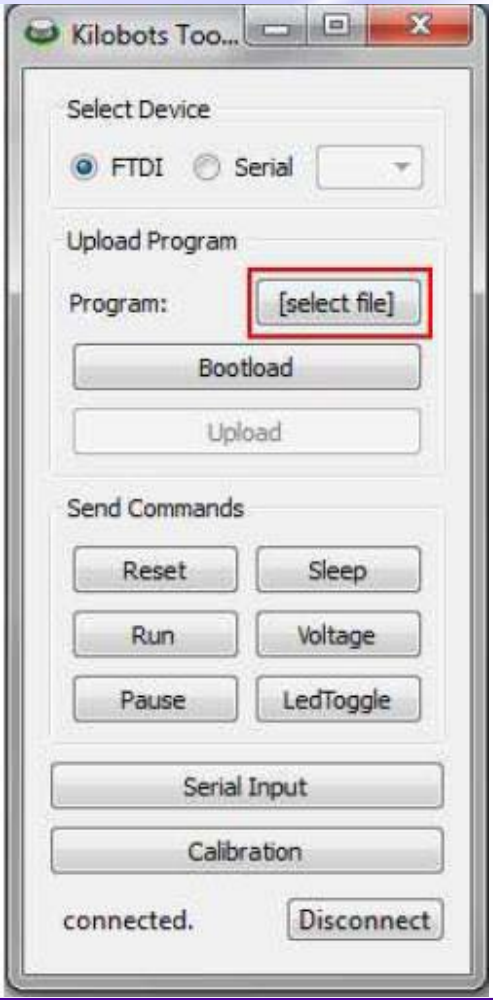
Kilobots should be operated on a smooth, flat surface to ensure proper robot mobility. To aid communication, the surface should be glossy or reflective.

The OHC controller should hung above the Kilobots at a distance of about one meter. The robots beneath the OHC will receive communications from the OHC.



Overhead Controller (OHC) Interface Overview - 1

Select the program to run



Kilobots into programming mode (blue led)

Program is uploaded (Kilobots are blinking)

Overhead Controller (OHC) Interface Overview - 2

Reset: Jump to the user program starting point, resetting all states, and stop. Remain idle (**blinking green**) waiting for the next command.

Run: Run the user program.

Pause: Pause the user program (preserves state, so that the program resumes where it left off).

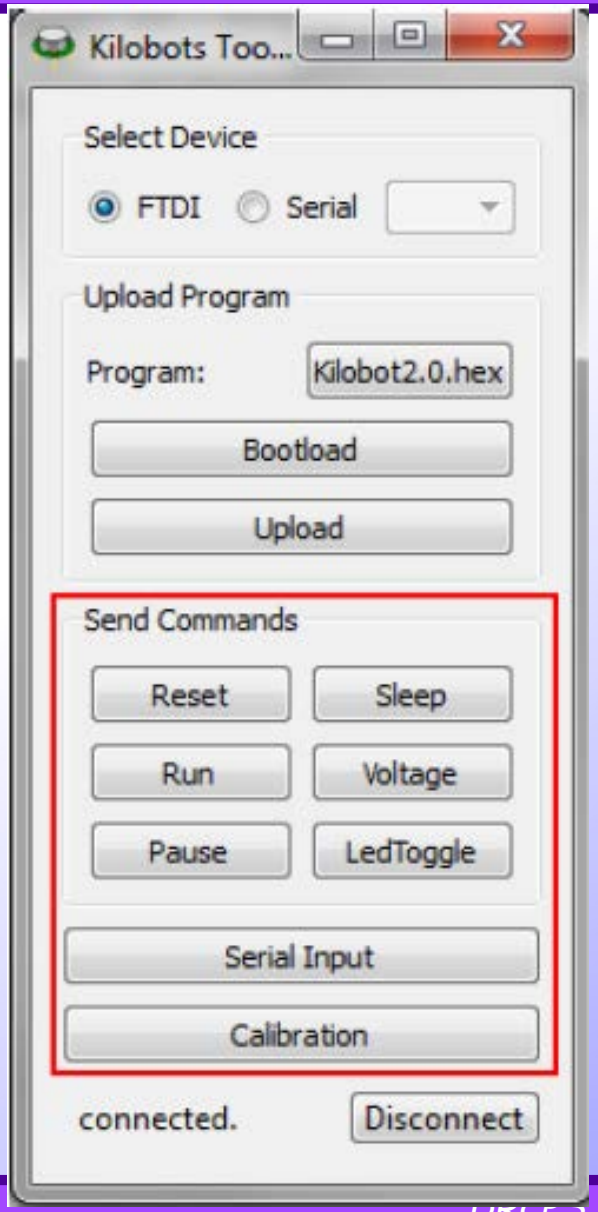
Sleep: Switch to low-power sleep mode (leds flash white once every 8 seconds).

Voltage: Display voltage level using LED (**blue/green** = charged, **yellow/red** = battery low)

LedToggle: Toggle LEDs on the controller, used to check communication between PC and controller

Serial Input: Show Kilobot messages using 2-wire serial cable.

Calibration: Set the UID of a Kilobot and their motor. (next slide)



Overhead Controller (OHC) Interface Overview - 2

Reset: Jump to the user program starting point, resetting all states, and stop. Remain idle (blinking green) waiting for the next co

Run: Ru

Pause: P
prograr

Sleep: S
every 8

Voltage
yellow/

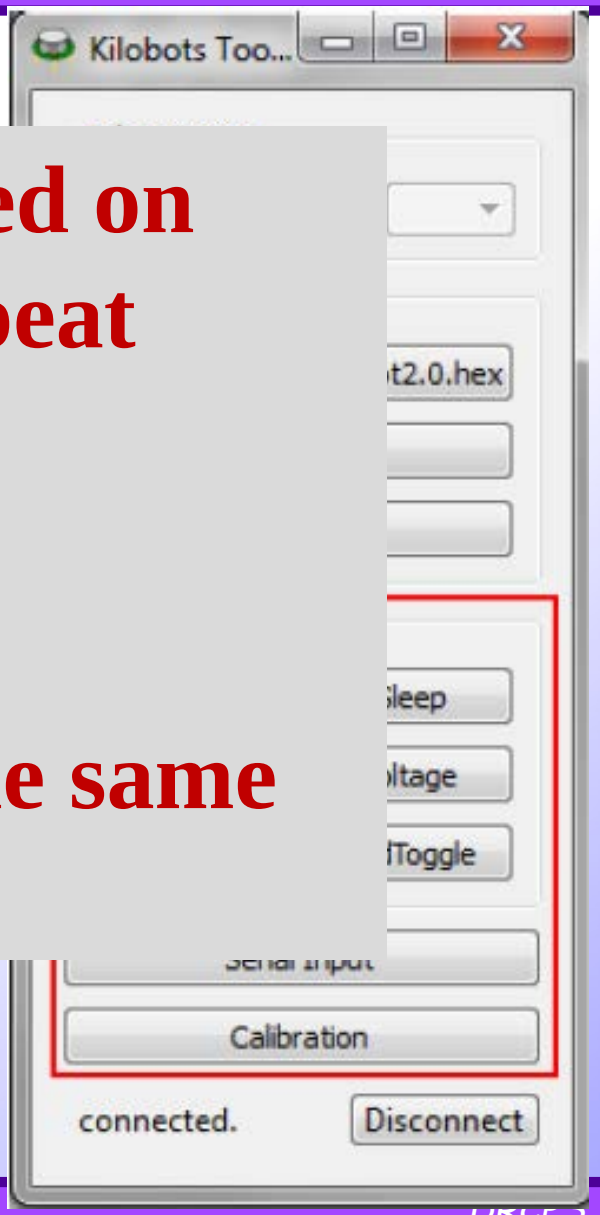
LedTog
commu

Serial

Calibration: Set the UID of a Kilobot and their motor. (next slide)

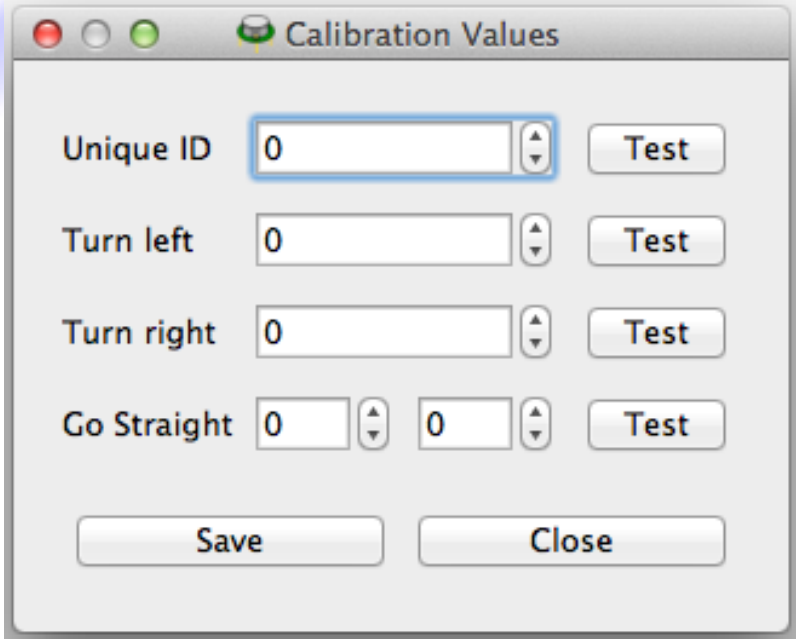
Note: once one button is pressed on KiloGUI, the action will be repeat infinitely.

To avoid any trouble with IR communication, press again the same button to stop the repeat.



Calibration

Open up the KiloGUI program and click on the Calibration button, you will be presented with the following screen:



NO VALUES ABOVE 100

1) Set **ONE** kilobot under the OHC
2) Open the calibration panel
3) Select a value for **turning left**, click test to tell the robot to move using this value. Values between 60 and 75 work best for turning. Follow the same procedure for **turn right**.
4) To calibrate to **go straight**, you can use the values you already found for turn left and turn right as a good initial guess. Usually go straight values should be smaller than the turning left and turning right values to achieve a good motion.
5) Choose a **unique ID** for your robot by typing an integer in the unique ID box, and clicking test.
6) Click **Save** to write these changes to the EEPROM ("permanent") memory of your robot.

Execution Example

1) Run ***kilogui.exe***, click the button ***[select a file]*** to browse and select your files.

2) Place *ALL* Kilobots in PAUSE mode (LED flashing green) underneath the OHC, and press the "***Bootload***" button in ***Kilogui.exe***. The Kilobots will turn their LED Blue to indicate that they are ready to be programmed.

3) Then press the ***"Upload"*** button to start programming. The Kilobots will flash Green and Blue alternatively.

4) Once the SW is uploaded, Kilobots return to PAUSE state (LED flashing green).

5) To run the program, press the "Run" button in *KiloGUI*.

6) Press the "Pause" button in *KiloGUI* to stop the program.

Note: once one button is pressed on KiloGUI, the action will be repeat infinitely. To avoid any trouble with IR communication, press again the same button to stop the repeat.

API - important recap - 1

At its core the Kilobot Library provides the function kilo_init() to initialize the hardware of the kilobots, and the function kilo_start() that uses a basic event loop programming paradigm (this relies on a setup and a loop callback, similar to those used in the arduino software).

The API also provides functions to read the various sensors available to the kilobots (get_ambientlight(), get_voltage(), get_temperature()), and also to control the individual pager motors and the RGB led present in each kilobot (set_motors(), set_color()).

API - important recap - 2

The user can **register callbacks to interact with the messaging subsystem**. There are callbacks for the events of message reception (kilo_message_rx), message transmission (kilo_message_tx), and notification of successful transmission (kilo_message_tx_success). By default **every kilobot attempts to send message twice per second**. Advanced users can modify this through the kilo_tx_period variable, although this is discouraged unless you know what you are doing.

To prevent collisions **the kilobot library uses a basic exponential back-off strategy with carrier sensing. There are no acknowledgement packets, and as such a message is considered to be successfully transmitted when a kilobot is able transmit a message without detecting any contention in the channel.**

FUNCTIONS

delay()
estimate_distance()
get_ambientlight()
get_temperature()
get_voltage()
kilo_init()
kilo_start()
message_crc()
rand_hard()
rand_seed()
rand_soft()
set_color()
set_motors()
spinup_motors()

API - https://www.kilobotics.com/docs/globals.html

delay()

Pauses the program for the specified amount of time.

This function receives as an argument a positive 16-bit integer ms that represents the number of milliseconds for which to pause the program.

estimate_distance()

Estimate distance in mm based on signal strength measurements.

This function receives as an argument the signal strength measurements d taken during message reception, and returns a positive integer that represents the distance estimate in **mm** towards the robot that originated the message.

API - https://www.kilobotics.com/docs/globals.html

get_ambientlight()

Read the amount of ambient light.

This function returns a 10-bit measurement (0 to 1023) that represents the amount of ambient light detected by the photo diode available in the kilobot. When the analog-to-digital converter is unavailable and the voltage cannot be measured, this function will return -1.

get_temperature()

Read the temperature of the kilobot.

This function returns a 10-bit measurement (0 to 1023) that represents the temperature **of the board** of the kilobot. This sensor is only capable of **detecting large temperature changes (in the order of 2 Celsius degrees or more)**. When the analog-to-digital converter is unavailable and the voltage cannot be measured, this function will return -1. As such, it is **only useful only to detect drastic changes in the operating environment of the kilobot**.

API - https://www.kilobotics.com/docs/globals.html

get_voltage()

Read the amount of battery voltage.

This function returns a 10-bit measurement (0 to 1023) that represents the amount of voltage that remains in the battery. It can be used to determine if the kilobot should be recharged. When the analog-to-digital converter is unavailable and the voltage cannot be measured, this function will return -1.

kilo_init()

Initialize kilobot hardware.

This function initializes all hardware of the kilobots. This includes calibrating the hardware oscillator, setting hardware timers, configuring ports, setting up analog-to-digital converters, registering system interrupts and the initializing the messaging subsystem.

It is recommended that you call this function **as early as possible inside the main function of your program.**

API - https://www.kilobotics.com/docs/globals.html

kilo_start()

Start kilobot event loop.

This function receives two parameters. The first parameter setup is a function which will be called once to perform any initialization required by your user program. The second parameter loop is a function that will be called repeatedly to perform any computations required by your user program.

Using the overhead controller it is possible to interrupt the event loop to trigger events such as program start/resume, program pause, and program restart.

Parameters

setup put your setup code here, to be run only once

loop put your main code here, will be run repeatedly

```
uint32_t counter;

void setup() {
   counter = 0;
}

void loop() {
   counter++;
}

int main() {
  kilo_init();
  kilo_start(setup, loop);
  return 0;
}
```

API - https://www.kilobotics.com/docs/globals.html

message_crc()

Function to compute the CRC of a message struct.

This function receives as input a pointer to a message structure msg, and uses the message data and type to compute and return a uint16_t CRC value.

Parameters

msg Pointer to an input message.

Returns

A 16 bit CRC of the message payload.

```
message_t msg;
msg.data[0] = 1;
msg.data[1] = 2;
...
msg.type = NORMAL;
msg.crc = message_crc(&msg);
```

rand_hard()

Hardware random number generator.

This function uses the analog-to-digital converter to generate an 8-bit random number. Specifically, the robot measures **its own battery voltage and extracts randomness from the least significant bit by using Von Neumann's fair-coin algorithm.** By its nature, this function is slow, use rand_soft() if you want a faster alternative, and you can seed the software random generator using the output of rand_hard().

API - https://www.kilobotics.com/docs/globals.html

rand_seed()

Seed software random number generator.

This function changes the seed used by the software random number generator implemented by rand_soft().

rand_soft()

Software random number generator.

This function implements a linear-shift-register to implement an 8-bit pseudo-random number generator. The seed of the random number generator can be controlled through rand_seed().

API - https://www.kilobotics.com/docs/globals.html

set_color()

Set the output of the RGB led.

This function receives an 8-bit unsigned integer whose bits are used to determine the output of the RGB led mounted on the kilobot. Each color has a 2-bit resolution which allows set each color channel independently from off (0) to full-brightness (3).

The convenience macro RGB can be used to set the individual bits. For instance RGB(0,0,0) turns all color channels off, and therefore the RGB led remains off. Meanwhile RGB(0,3,0) turns the green channel to full intensity and turns all other channels off, which results in an RGB led displaying a bright green color.

```
// blink dim RED once per second
while (1) {
  set_color(RGB(1,0,0));
  delay(500);
  set_color(RGB(0,0,0));
  delay(500);
}
```

API - https://www.kilobotics.com/docs/globals.html

set_motors()

Set the power of each motor. The power received by the left and right motor is controlled using hardware pulse-width-modulation (PWM) and can be set using this function.

The parameter left and right are 8-bit unsigned integers (0 to 255) that control the duty-cycle of the PWM signal from 0% to 100%. In other words, setting a motor to 0% duty-cycle equates to running off the motor, and setting a motor to 100% duty-cycle equates to running the motor at full power. For the most part, motors should only be set to the calibrated values kilo_turn_left, kilo_turn_right, kilo_straight_left and kilo_straight_right. When a motor transitions from being off (0% duty cycle) to being on (> 10% duty cycle) it must first be turned on at full-speed for 15ms to overcome the effects of static friction.

In a 2 second interval no motor at 100% duty-cycle (255) for more than 50 ms at a time!!!!!!!

```
// turn motors off
set_motors(0, 0);
// spin up left motor for 15ms
set_motors(255, 0);
delay(15);
// turn the kilobot left for 2 seconds
set_motors(kilo_turn_left, 0);
delay(2000);
// go straight for 2 seconds
set_motors(kilo_straight_left, kilo_straight_right);
delay(2000);
```

API - https://www.kilobotics.com/docs/globals.html

spinup_motors()

Turn motors at full-speed for 15ms.

When the robot transitions from being stationary (motors off) to being mobile (one or both motors on) it must overcome the effects of static friction. For that purpose, the motors can be turned-pn at full-speed during 15ms. This function does precisely that, and is equivalent to the following code:

```
set_motors(255, 255);
delay(15);
```

Note

Observe that the spinup() function turns both motors on. In some cases (when turning left or turning right) this is not required, and thus to achieve smoother motion you can do manual spinup of a motor. See set_motors() for an example.

In a 2 second interval no motor at 100% duty-cycle (255) for more than 50 ms at a time!!!!!!!

VARIABLES

kilo_message_rx
kilo_message_tx
kilo_message_tx_success
kilo_straight_left
kilo_straight_right
kilo_ticks
kilo_turn_left
kilo_turn_right
kilo_uid

Variables - https://www.kilobotics.com/docs/globals_vars.html

kilo_message_rx

Callback for message reception.

This callback is triggered every time a message is successfully decoded. The callback receives two parameters, a pointer to the message decoded, and a pointer to the distance measurements from the message originator.

Note
You must register a message callback before calling kilo_start.

```
uint8_t recvd_message = 0;

// receive message callback
void rx_message(message_t *msg, distance_measurement_t *d) {
    recvd_message = 1;
}

void setup() {
    recvd_message = 0;
}

// blink green when a new message is received
void loop() {
    if ( recvd_message ) {
        recvd_message = 0;
        set_color(RGB(0,1,0));
        delay(100);
        set_color(RGB(0,0,0));
    }
}

int main() {
    kilo_init();
    // register message callback
    kilo_message_rx = rx_message;
    kilo_start(setup, loop);

    return 0;
}
```

Variables - https://www.kilobotics.com/docs/globals_vars.html

kilo_message_tx

Callback for message transmission.

This callback is triggered every time a message is scheduled for transmission (roughly twice every second). This callback returns a pointer to the message that should be sent; if the pointer is null, then no message is sent.

```
message_t msg;
uint8_t sent_message;

// message transmission callback
message_t *tx_message() {
    return &msg;
}

// successful transmission callback
void tx_message_success() {
    sent_message = 1;
}

void setup() {
   msg.type = NORMAL;
   msg.crc = message_crc(&msg);
}

// blink red when a new message is sent
void loop() {
    if ( sent_message ) {
        sent_message = 0;
        set_color(RGB(1,0,0));
        delay(100);
        set_color(RGB(0,0,0));

    }
}

int main() {
    kilo_init();
    // register message transmission calback
    kilo_message_tx = tx_message;
    // register tranmsission success callback
    kilo_message_tx_success = tx_message_success;
    kilo_start(setup, loop);

    return 0;
}
```

Variables - https://www.kilobotics.com/docs/globals_vars.html

kilo_message_tx_success

Callback for successful message transmission. This callback is triggered every time a message is sent successfully. It receives no parameters and returns no values.

Warning

The message subsystem has no acknowledgements, therefore successful message reception is not guaranteed. Instead the successful message callback is called when a message is transmitted and no contention is detected on the channel.

kilo_straight_left

Calibrated straight (left motor) duty-cycle. This variable holds an 8-bit positive integer which is the calibrated duty-cycle used for the left motor to go straight. This must be used in conjunction with kilo_straight_right.

kilo_straight_right

Calibrated straight (right motor) duty-cycle. This variable holds an 8-bit positive integer which is the calibrated duty-cycle used for the right motor to go straight. This must be used in conjuction with kilo_straight_left.

Variables - https://www.kilobotics.com/docs/globals_vars.html

kilo_ticks

Kilobot clock variable.

This variable holds a 32-bit unsigned positive integer. This variable is initialized to zero whenever the program run at the kilobot is reset (or when the kilobot is first turned on). It is incremented approximately 32 times per second, or once every 30 ms.

```
void setup() {
    last_changed = kilo_ticks;
}

// blink the LED green for 50ms, once every 2 seconds.
void loop() {
    if (kilo_ticks > last_changed + 64) {
        last_changed = kilo_ticks;
        set_color(RGB(0,1,0));
        delay(50);
        set_color(RGB(0,0,0));
    }
}

int main() {
    kilo_init();
    kilo_start(setup, loop);

    return 0;
}
```

Variables - https://www.kilobotics.com/docs/globals_vars.html

kilo_turn_left

Calibrated turn left duty-cycle.

This variable holds an 8-bit positive integer which is the calibrated duty-cycle for turning left.

kilo_turn_right

Calibrated turn right duty-cycle.

This variable holds an 8-bit positive integer which is the calibrated duty-cycle for turning right.

kilo_uid

Kilobot unique identifier.

This variable holds a 16-bit positive integer which is designated as the kilobot's unique identifier during calibration.

MACROS

debug_init

This function initializes the hardware serial communication of the Kilobot. This function must be called before the kilo_start() function, but after the kilo_init() function.

Moreover, for the serial to be enabled the variable DEBUG (notice capitalization) must be defined before debug.h is included. Thereafter, the printf() function can be used to transmit debugging information to the kilobot controller through the attached serial cable.

DATA STRUCTURES

distance_measurement_t

Every time a message is received by a kilobot, it collects **two 10 bit measurements** that represent the signal strength of the received message after going through an operational amplifier. This data structure stores these two measurements.

Using these two measurements it is possible to estimate the distance of the sender.

message_t

A message structure is 12 bytes in length and is composed of three parts: the payload (9 bytes), the message type (1 byte), and a CRC (2 bytes).

EXERCISE 1

BLINKY

- **Program**: Blink LEDS Red then Blue for 500ms each
- **Objective**: Introduce basic code structure (i.e. setup and loop) and basic functions such as set_color and delay
- **Code**: blink_led.c

Just play with the kilobots: the code is in the next slide.

The program will be run repeatedly until you either reset or pause the robot.

```
#include <kilolib.h>
void setup()
{
   // Put any setup code here. This is run once before entering the loop.
}

void loop()
{
   // Put the main code here. This is run repeatedly.

   // Set the LED red.
   set_color(RGB(1, 0, 0));
   // Wait half a second (500 ms).
   delay(500);
   // Set the LED blue.
   set_color(RGB(0, 0, 1));
   // Wait half a second (500 ms).
   delay(500);
}
```

```
int main()
{
   // Initialize the hardware.
   kilo_init();
   // Register the program.
   kilo_start(setup, loop);

   return 0;
}
```

EXERCISE 2

MOVEMENT

- **Program**: Move forward 2 sec, clockwise 2 sec, anticlockwise 2 sec, stop 5 sec, and repeat
- **Objective**: Introduce set_motors and calibration constants kilo_turn_left, kilo_turn_right, kilo_straight_left, kilo_straight_right
- **Code**: simple_movement.c

In this lab we will make ONE Kilobot go through its motions --forward, turn left, turn right-- in a loop. We will use the function set_motors that takes values for each of the two motors, and we will use the calibrated constants.

There's one more important thing you need to do, which is spinup_motors. When the motors are first turned on, we must set the motors to the maximum speed for 15 milliseconds or so, in order for the kilobot to overcome static friction. We call this spinning up the motors. Therefore, for the robot to move it must first spin up the motors and then set its desired motion. **Motors need to be spinup every time the robot changes its direction of motion**. This can be done through the spinup_motors function, also described in the API page.

```
#include <kilolib.h>

void setup()
{
}

void loop()
{
  // Set the LED green.
  set_color(RGB(0, 1, 0));
  // Spinup the motors to overcome friction.
  spinup_motors();
  // Move straight for 2 seconds (2000 ms).
  set_motors(kilo_straight_left, kilo_straight_right);
  delay(2000);

  // Set the LED red.
  set_color(RGB(1, 0, 0));
  // Spinup the motors to overcome friction.
  spinup_motors();
  // Turn left for 2 seconds (2000 ms).
  set_motors(kilo_turn_left, 0);
  delay(2000);

  // Set the LED blue.
  set_color(RGB(0, 0, 1));
  // Spinup the motors to overcome friction.
  spinup_motors();
  // Turn right for 2 seconds (2000 ms).
  set_motors(0, kilo_turn_right);
  delay(2000);

  // Set the LED off.
  set_color(RGB(0, 0, 0));
  // Stop for half a second (500 ms).
  set_motors(0, 0);
  delay(500);
}

int main()
{
  kilo_init();
  kilo_start(setup, loop);

  return 0;
}
```

EXERCISE 3 - DISPERSE

Program: Create a single robot that both sends and receives messages. The robot should check every second if it has received a message. If so, then it should pick a random direction to move in (50% straight, 25% left, 25% right) and set its led according to the motion (green=forward, red=left, blue=right). If not, it should stop and set its led white.

Objective: Put communication and motion together, avoid the delay() that is blocking, introduce rand_hard(), and create subroutine for cleaner and more efficient motion code.

See and execute code: disperse.c

EXERCISE 4 - COMPUTE DISTANCE

Modify the previous program for kilobots computing distance: they do not have to crash! Each kilobot continuously computes distance from the nearby kilobots:

- When distance is below 40 mm, LED is set to RED and Kilobot stops moving
- Otherwise, LED is set to BLUE

(here you need to develop your own code)

How to compute distance: you can check orbit_planet.c

EXERCISE 4 - How to understand Kilobot ID

- Assign an ID to 3 Kilobots (with values 0, 1, 2). This must be done through the configuration panel.
- Kilobot LED is set differently according to their ID:
  - Red (ID = 0)
  - Green (ID = 1)
  - Blue (ID = 2)
- After 5 seconds from start, Kilobot 0 chooses a random COLOR between R, G, B (use: 0, 1, 2) and transmits its decision to the other 2 Kilobots
- All kilobots set their LED to the selected color