# Modular Robots: from simulation to real world

*Frédéric Lassabe*

# Who am I ?

- PhD from DISC/OMNI in 2009 (LIFC at the time)
- MSc Distributed Systems in 2004 (UFC)
- UTBM Engineer in 2003
- Member of OMNI research team
- Assistant professor in UTBM
- Main research topics :
  - Programmable matter : distributed algorithms
  - Resources optimisation
  - Mobility analysis
  - Project leader on Blinky Blocks and geologic
- Main teaching topics :
  - Networks
  - Linux administration and programming
  - Indoor positioning

# Why Blinky Blocks

- Programmable matter
  - Hardware challenges (size reduction, energy, movements, etc.)
  - Software challenges : embedded distributed systems
- Blinky Block : static prototype of programmable matter basic element
  - No actuator
  - No battery
  - Reasonable size (on the shelf components)
  - Serial communication : real life execution of distributed algorithms
- This session
  - Blinky Blocks hardware and software architectures presentation
  - "Administration" with BB bootloader
  - Development and execution of BB applications

First contact with Blinky Blocks

- Hardware
- Required accessories
- BB bootloader and applications

Bootloader administration

Programming applications for BB

First contact with Blinky Blocks

- Hardware
- Required accessories
- BB bootloader and applications

Bootloader administration

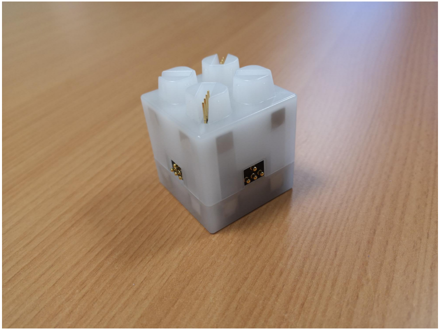
Programming applications for BB

# Main characteristics

- A Blinky Block = a complete system
- Remarkable components :
  - Microcontrolleur (ARM Cortex M0)
  - Memories (flash 128 KB, SRAM 32 KB)
  - Multicolor LED
  - Microphone
  - Accelerometer
  - Gyroscope
  - Buzzer
  - **6 serial network interfaces** (one for each side)
- Able to send and receive *stimuli*

# BB external aspects

- 4cm wide cube, a bit similar to a big Lego brick
- Connectors (power supply and UART) on each side



# Top/Bottom connectors

- Top and bottom sides can assemble in 4 possible directions
- To provide connection whatever the direction :
  - One UART has two physical connectors (top and bottom sides)
  - Top : diagonal
  - Bottom : aligned
- Relative orientation between stacked (i.e. top-bottom connection) unknown

First contact with Blinky Blocks

Hardware

Required accessories

BB bootloader and applications

Bootloader administration

Programming applications for BB

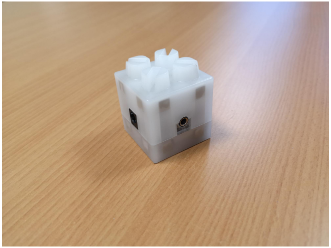
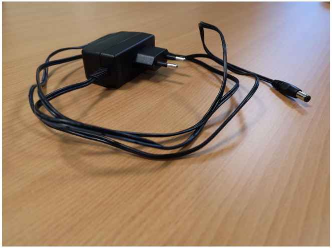
# Power supply

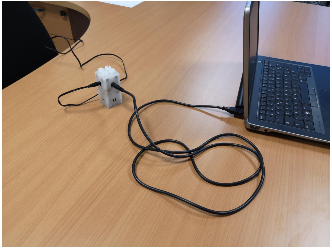
- In order to operate, a BB requires
  - Power supply
  - Has no internal battery
  - $\Rightarrow$ cable power supply
- Propagated power supply
  - One 5V PS
  - Several BB are powered
  - Current is propagating through sides connectors
  - Uses a dummy BB with a PS connector (this cube is named **programming block**)

# Connecting to a computer

- Interactions with BB require
  - A transmission medium
    - USB/serial cable
    - Through a second dedicated connector on the programming block
  - A computer
    - With one available USB port
    - BB controlling software : blinkyApploaderCLI

# Accessories and full setup



First contact with Blinky Blocks

- Hardware
- Required accessories
- BB bootloader and applications

Bootloader administration

Programming applications for BB

# BB Software architecture

- Programs are stored in flash memory
- No OS, task management, etc.
- 2 programs :
  - Bootloader
    - Manages BB startup and application launching
    - Relies on a distributed programming protocol (BBPP)
    - Can execute commands for BB administration
    - Set at address 0x8000000 (start of flash)
  - Application
    - Written by users (i.e. You)
    - A given application goal : coordinates calculation, artistic performances, etc.
    - Set at address 0x8010000 (Second half of flash)

# Writing an application

- C language (STM32 has C++ support but our API doesn't)
- ARM toolchain (architecture is different from x86/x86_64)
- Environment to develop and compile
- STM32CubeIDE
  - Provided by STMicroelectronics (brand of BB MCU)
  - Manages hardware configuration
  - Manages STMicroElectronics chips SDK's
  - Generates code templates to be completed by the developer
- For BB : Empty project with BB application template code
- BB tutorial virtual machine : everything already in-place

# Starting an application

- May be automatic (see configuration)
- Or use a command
- Bootloader calls application's main function
- Reinitializes MCU
- Go back to bootloader : restart BB
  - Hard reboot : unplug PS and plug back
  - or Reboot command

First contact with Blinky Blocks

Bootloader administration

Programming applications for BB

# Bootloader

- Bootloader manages BB and applications deployment
- Allows a set of commands
  - Changing colors
  - Emit sounds
  - Read configuration
  - Jump to the application
  - Modify persistent configuration
    - Set of variables
    - Stored in flash (address 0x8007800)
  - Deploy an application
  - Build a spanning tree
  - Reboot
- Commands are issued from the computer by blinkyApploaderCLI

# blinkyApploaderCLI options

- A couple interesting options from blinkyApploaderCLI
  - -q quit blinkyApploaderCLI after completion of all the commands queue
  - -t Creates a spanning tree on BB ensemble. All subsequent commands apply to the complete spanning tree
  - -s Specifies name/path of serial port connection (/dev/ttyUSB[ :digit :] on Linux, COM[ :digit :] on Windows
- Commands can be chained, i.e.

  ```
  blinkyApploaderCLI -t -p program.hex -q
  ```

  is valid.

# Color change

- **-c** command followed by RGB color code (separated by columns, no spaces)
- Changes color of target BB
- For instance :
  - blinkyApploaderCLI -c 5,0,0 lights connected BB in dark red
  - blinkyApploaderCLI -t -c 0,0,5 lights all connected BB in dark blue
- Useful to test connections and spanning tree

# Software identifiers allocation

- **-k** command followed by value of the first ID to be set
- First ID set on the directly connected BB
- On some algorithms, a leader is required
- Can be chosen by an election (prior phase)
- To simplify : set IDs and arbitrarily fix the leader (e.g. ID=1)
- Required a spanning tree
- Returns to computer the highest ID allocated
- Example :

  ```
  blinkyApploaderCLI -t -k 1
  ```

  allocates IDs starting from 1

# Application deployment

- **-p** command followed by the name of the application file
- Application must be provided as an Intel HEX file
- Deployment requires a spanning tree
- BB go through verious states (validation, initialization, propagation) before programming.
- Computer shows a programming progress value (from 0 to 100%).
- BB switch to dark blue when programming is finished.
- Example

  `blinkyApploaderCLI -t -p application.hex`

  deploys application from file application.hex

# Application execution

- **-j** command followed by application address (currently always 0x8010000)
- Commands the bootloader to run this application
- Example

  ```
  blinkyApploaderCLI -j 0x8010000
  ```

  starts application located at address 0x8010000
- If address in incorrect or no application is deployed, BB crashes and must be hard-reset.

First contact with Blinky Blocks

Bootloader administration

Programming applications for BB
- LED
- Time
- Sound

# Basic principle

- Do not use STM32CubeIDE code generator (hence, do not tinker with project .ioc file)
- Use application template with all BB basic features
  - Hardware initialisation
  - BB network protocol
  - A subset of bootloader commands
- BB tutorial VM : project named *application*
- Relies on a precompiled and configured library

# Init/loop

- Similar to Arduino :
  - BBinit function (executed once on startup)
  - BBloop function (executed repeatedly between system functions)
- No network polling (this part differs from Arduino) :
  - API exposes a user packet callback to be implemented
  - User manages its packets payload and identification
- Signatures and definitions of functions are located in user_code .h et .c files

First contact with Blinky Blocks

Bootloader administration

Programming applications for BB

- LED
- Time
- Sound

# Light a color by its name

- Uses named colors as an enum

```
#include <light.h>
// light.h defines colors, such as:
typedef enum __packed {
        RED, ORANGE, YELLOW, GREEN, CYAN,
        BLUE, PURPLE, GREY, WHITE, DARK_RED,
        /*... ,*/ NB_COLORS
} ColorName;
// ...
setColor(uint8_t color_index); // color index < NB_COLO
// Example:
setColor(DARK_RED); //Sets BB LED to dark red (i.e. 20,
```

# Setting color with RGB code

- If no preset color suits you
- You want colors depending on variables
- Colors set by RGB (red, green, blue)
- Each value from 0 to 100 (greater values are capped)

```
#include <abstraction.h>

set_RGB(20, 0, 20); // Will set BB LED to a dark purple
```

First contact with Blinky Blocks

Bootloader administration

Programming applications for BB

- LED
- Time
- Sound

# Getting system time

- HAL_GetTick() function (returns milliseconds since program startup)
- Used to compute durations :
  - Initialize start date : call HAL_GetTick()
  - Do something
  - Call HAL_GetTick()
  - Difference between both dates is the duration
- Useful to trigger events after a given delay.

# Example to trigger a function after delay

```
#include <stm32f0xx_hal.h>
uint32_t start_time;
void BBinit() {
        start_time = HAL_GetTick();
}

void loop() {
        if (HAL_GetTick() - start_time < 250) { // 250ms
                // Do some init and wait for updates
        } else {
                setColor(GREEN);
        }
}
```

# Waiting

- HAL_Delay(msec) function : DO NOT USE, BLOCKING FUNCTION !
- Use HAL_GetTick() to compute a delay
- Or use sleep_sec et sleep_msec (wait for some seconds or milliseconds)
  - Similar to HAL_Delay but :
  - non blocking (messages and other events still are processed)
  - less precise (some processing may end up with a slightly longer time)

First contact with Blinky Blocks

Bootloader administration

Programming applications for BB

- LED
- Time
- Sound

# Function for using sound

- BB have a buzzer
- Emit a given frequency for a given time

```c
#include <abstraction.h>

void make_sound(unsigned short usFreq, \
                unsigned short usDuration);
```

- With :
  - usFreq is the frequence in Hz
  - usDuration is the sound duration in milliseconds
- Warning : function doesn't block during sound

# Example : a bipping application

```
#include <hwBuzzer.h>

void BBloop() {
  static uint32_t next_change=0;
  uint32_t current_time=HAL_GetTick();
  if (current_time >= next_change) {
    make_sound(440, 500); // Play LA
    next_change = current_time + 1500;
  }
}
```

Thanks for your attention !