Internet of Toys: Julia and Lego Mindstorms

Robin Deits

December 13, 2015

Abstract

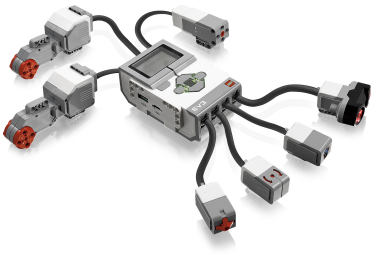
The latest version of the Lego Mindstorms robotics system is an excellent candidate for the exploration of distributed robotics. I implemented bindings to the `ev3dev` operating system, which runs on the Mindstorms EV3 brick, in Julia. Using those bindings, I constructed a library to perform a simple cooperative mapping task on a pair of mobile robots. Due to the hardware limitations of the EV3 processor, I was not yet able to run Julia onboard. Instead, I developed a simple server-client architecture using ZeroMQ [1] to allow Julia code to run off-board and control the Mindstorms robot over WiFi. With this system, I was able to map simple environments both in serial (with one robot) and in parallel (with a team of two robots).

Contents

1 Background

The Lego Mindstorms system was first released in 1998, with the Mindstorms Robotics Invention System [2]. The Mindstorms system consists of a central computer with input and output ports, and a set of lego-compatible motors and sensors, as seen in Fig. 1. Lego provides a development environment for the Mindstorms computer, in which users can create programs using a visual block language. There have been a variety of attempts to introduce more traditional programming languages to the Mindstorms environment, including Not Quite C [3] and Next Byte Codes [4]. These efforts have focused on writing and compiling code on a user's PC, and then deploying that code to the Mindstorms computer for execution, since the software on the Mindstorms brick was proprietary and closed.

(a) The original 1998 Mindstorms RCX, with sensors and motors attached.



(b) The 2013 Mindstorms EV3, with motors and sensors.

Figure 1: The Lego Mindstorms computer bricks, with attached peripherals. Figures reproduced from [5].

Fortunately, the situation has been radically improved with the newest Lego Mindstorms release, EV3. With EV3, the brick (the box which houses

the battery, computer, screen, and interface ports) is now a small Linux computer complete with USB and MicroSD ports. This means that it is now possible to install and even develop software directly on the EV3 using standard Linux tools. Even better, the EV3 is capable of booting directly from the MicroSD port, which opens the door for custom operating systems.

I have built my work on top of ev3dev, a custom Debian Linux distribution built specifically to run on the EV3. The ev3dev distro makes communicating with the basic EV3 hardware particularly easy by mapping sensors and motors directly to nodes in the filesystem. For example, instructing a motor to run continuously is as simple as:

```
echo run-forever > /sys/class/tacho-motor/motor2/command
```

The developers of ev3dev also provide higher level bindings for C++, Python, Lua, and node.js. Rather than building Julia wrappers for any of these language bindings, I have written a new set of bindings directly on top of the filesystem interface provided by the ev3dev OS.

2 Hardware Interface

2.1 Running Julia on the EV3

Creating a working build of Julia on the EV3 proved to be much more difficult than anticipated, and I was not able to cross-compile a usable version. The EV3 has an older ARM926EJ-S processor running the `ARMv5` instruction set. Although Julia has been successfully built on other ARM devices like the Raspberry Pi, the EV3 is more limited in three ways:

1. Instruction set: The EV3 uses the older ARMv5 instruction set. Julia has been built on systems using ARMv6 and ARMv7, but never yet on v5 [6].
2. Floating-point support: The EV3 processor has no hardware floating-point support, and instead relies on the compiler to emulate floating-point operations in software. Julia does not have built-in support for so-called "soft floating-point" CPUs.
3. Available RAM: The EV3 has only 64 MB of RAM, which may be enough to run Julia, but is not enough to compile its dependencies (particularly LLVM).

Since the resources on the EV3 were limited to a 300 MHz processor and 64 MB of RAM, I was forced to resort to cross-compilation to build Julia. Using the Brickstrap toolchain [7], I was able to build a complete image of the `ev3dev` operating system and compile Julia within Brickstrap but was never able to run the resulting Julia build without segmentation faults on startup. More discussion of the specific technical issues encountered can be seen in the discussion on the `julia-dev` mailing list in [8].

Despite these setbacks, it is not yet clear that Julia cannot be built on the EV3. I plan to continue experimenting with Julia, specifically by attempting to identify and remove components of Julia which are failing on the EV3.

2.2 Running Julia Off-board

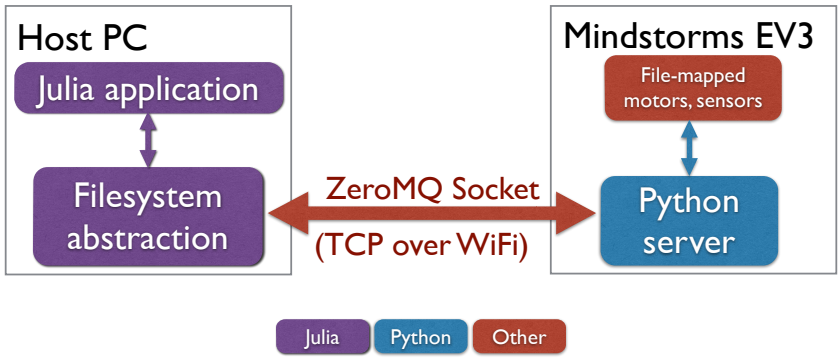


Figure 2: System architecture for controlling the Mindstorms EV3 from a separate computer in Julia. ZeroMQ is used as a communication layer over WiFi to emulate the process of running Julia directly on the EV3

Without a working Julia build on the EV3, I decided to create a minimal system to allow Julia to run on an off-board PC while still communicating with the EV3's sensors and motors. The easiest way to do this would be to mount the virtual file system provided by `ev3dev` on the user's PC, using a tool like AFS or SSHFS. I implemented the SSHFS mount, but experienced very poor reliability, with many commands being dropped or intermittently ignored. It appeared that the unreliable WiFi connection to the EV3 made mounting a stable remote filesystem too difficult.

Instead, I chose to use ZeroMQ [1], a robust, high-performance communication library to transmit messages between the EV3 and the host PC. In all of my experiments, ZeroMQ was remarkably reliable, surviving network dropouts and various software crashes without issue. The structure of the system can be seen in Fig. 2. On the EV3, a small Python program acts as a server using a ZeroMQ socket. The Python server listens for messages containing one of three commands: `r`, `w`, and `l` to read, write, or list, respectively, files on the device. Since all motors and sensors are mapped to files in `ev3dev`, those commands are all that are needed for full remote control of the EV3. The Python server also caches the handles to these virtual files in order to allow faster access to the hardware devices.

In Julia, a few helper classes abstract away the ZeroMQ layer in order to make running off-board similar to running directly on the EV3. The `AbstractNode` type represents a file or directory on a local or remote filesystem. Its subtypes, `LocalNode` and `RemoteNode` contain the implementation details for a particular real file. In the case of a `LocalNode`, only the path to the file is stored. A `RemoteNode`, on the other hand, stores its file path, the hostname or IP address of the EV3, and a ZeroMQ socket for communication with that host. The `Motor` and `Sensor` types

require only an `AbstractNode` to handle reading and writing of data, so they can be used without regard for whether Julia is being run on the EV3 or on a separate PC.

3 Software

3.1 Low-Level Bindings

At the lowest level of the Julia code that I developed for this project are the basic input and output bindings for EV3 sensors and motors. Every motor or sensor provides many possible signals and outputs, including the current sensor value, motor position, driver name, firmware version, physical port name, and calibration information. Through the `AbstractNode` interface, it is easy to read input data (as strings) and write commands (as strings), but this presents the user with a complex, unregulated API. For example, to command a motor to run continuously, a user might write directly to the abstract node:

```
write(command_node, "run-forever")
```

or to read a sensor value, the user might write:

```
value = parse(Int, read(value_node))
```

but this allows users to send potentially illegal commands to the motors, and requires that the user understand the type of data read from the sensor's value file.

Instead, to create a safer, easier API, the existing EV3 language bindings create dedicated functions or class methods to read and/or write each of the available signals. However, this results in a great deal of boilerplate code. The `ev3dev` Python bindings resolve the issue by using Liquid, a templating language, to generate Python code in a separate build step. But using a build system to generate code removes some of the advantages of a high-level, interpreted language like Python.

Julia's metaprogramming support, on the other hand, makes it much easier to generate all the necessary boilerplate directly within the language. I created three macros, `@readable`, `@writeable`, and `@readwriteable` which take a description of a signal, a category of devices which support it, and functions to parse outputs and/or validate inputs. For example, to declare that `port_name` is a readable attribute of all devices, we write:

```
@readable port_name AbstractDevice as_string
```

which produces the following code (after some cleanup for readability):

```
function port_name(dev::AbstractDevice)
    as_string(read(dev,"port_name"))
end
```

Likewise, to declare that `command` is a readable and writeable attribute of all Motors, we write:

```
@readwriteable command Motor as_string x->in(x, valid_commands)
```

which produces a pair of functions to read and write commands:

```
function command(dev::Motor,value)
    if !(((x->begin in(x,valid_commands) end))(value))
        error("Validation function: ",
              (x->begin in(x,valid_commands) end),
              " failed with value: ",value)
    end
    write(dev,"command",string(value))
end

function command(dev::Motor)
    as_string(read(dev,"command"))
end
```

This structure makes writing safe input and output methods very easy with minimal boilerplate. In the future, as I add support for more detailed categories of devices, this approach should also make is easy to create only the appropriate functions for a given device. For example, we may choose to break up the Motor class into motors which provide position sensors (and thus have an `@readable position`) and those which do not. This is easy to indicate by specializing the class which we pass into the `@readable` macro.

4 Collaborative Mapping on the EV3

As a demonstration of the low-level language bindings, I created a simple parallel mapping project, in which a team of two robots explore the world simultaneously, gathering point cloud representations of their local areas.

4.1 Hardware Design

The hardware used for the mapping project is shown in Fig. 3. The design is based on plans provided by Lego for a simple two-wheeled robot, but with the addition of a rotating ultrasound sensor head, modifications to the gyroscope attachment, and a guard for the external USB WiFi module (not shown). The Lego motors for the drive wheels and the head provide absolute position encoders, and the gyroscope provides a low-drift orientation estimate. The ultrasound sensor provides an estimate of the distance from the sensor to the nearest hard surface, although its angular resolution is limited to approximately 20 degrees.

4.2 Software Design

The software developed for this project can be found in three small software projects: `Ev3.jl` (low-level bindings), `Behaviors.jl` (finite-state machine behaviors), and `MappingRobots.jl` (state-estimation and point-cloud gathering). All are available on Github at github.com/rdeits/Ev3.jl, github.com/rdeits/Behaviors.jl, and github.com/rdeits/MappingRobots.jl.

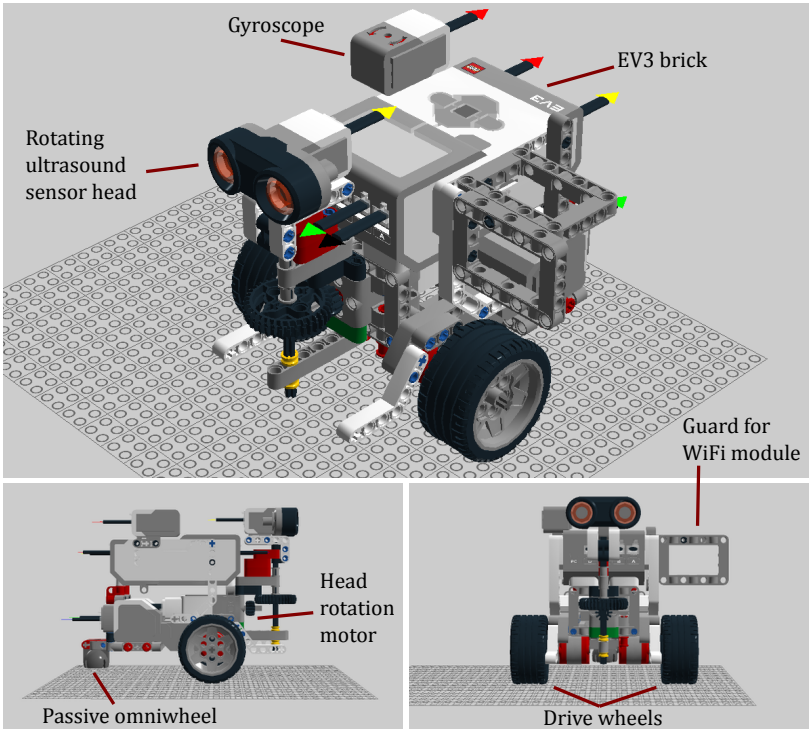


Figure 3: The Lego robot used for the collaborative mapping project, rendered using the Lego Digital Designer software [9].

4.2.1 Finite-State Machine Behaviors

The mapping robot is controlled by a set of behaviors and acts as a finite state machine. Each behavior consists of an action—a function which can be applied to the robot's motors—and a set of transitions which describe conditions for switching to another behavior. Multiple behaviors can be active simultaneously, and all active behaviors apply their individual actions to the robot's state. Only four behaviors and a start and halt state are needed to define the entire mapping task, as shown in Fig. 4. The general framework for defining behaviors, actions, and transitions is implemented in a standalone library, `Behaviors.jl`, created for this project.

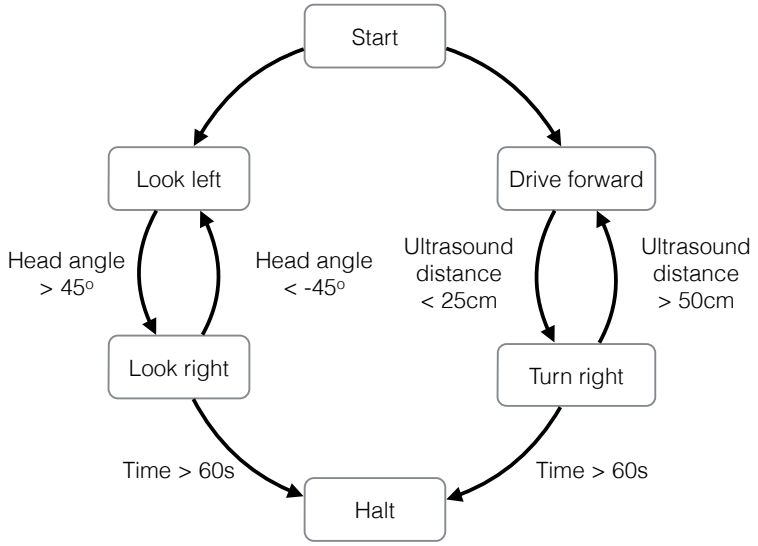


Figure 4: The behaviors used by each robot in order to explore its environment, implemented in `Behaviors.jl`. Additional timeout transitions from `Look left` and `Drive forward` to `Halt` omitted for clarity

.

4.2.2 Mapping

The actual mapping code is a small set of functions built on top of the low-level bindings and the general behavior engine. At each iteration of the control loop, the code gathers data from the ultrasound and gyroscope sensors and records the absolute position of the drive and head rotation motors. That input is passed into the behavior engine, which resolves any necessary behavior transitions. The active behaviors then act on the robot's motors, causing it to drive and look in the appropriate directions to avoid obstacles and scan the environment. In addition, the change in the position of the drive wheels and the orientation of the gyroscope is

used to update a dead-reckoning estimate of the robot's current position in the world.

When the ultrasound detects a nearby object (less than 2 m away), a single point is added to the robot's map. Over the course of exploring an environment, the robot gathers an increasingly dense pointcloud.

4.2.3 Parallel Collaborative Mapping

Using Julia's built-in support for parallel computing, running multiple robots simultaneously is not much more difficult than simply running one robot. In this simple task, the robots need not share any information as they explore. Instead, each robot is initialized from a known initial position, and then it returns its entire pointcloud to the master process when its mapping behavior is complete. A collaborative mapping process, in which the robots share data to improve their maps and localization, would be a very interesting future area of work, but was beyond the scope of this project.

Running two robots in parallel and concatenating their results is as simple as:

```
addprocs(2)
@everywhere hostnames = ["192.168.1.27", "192.168.1.25"]

maps = @sync @parallel (vcat) for i = 1:2
    robot = construct_robot(hostnames[i])
    run_mapping(robot)
end
```

5 Example Results

5.1 Mapping

Using the complete Julia mapping stack, running on a host PC and communicating with the EV3 over ZeroMQ, I was able to produce approximate maps of a set of artificial environments. The robot, built from the model shown in Fig. 3 can be seen in Fig. 5.

Using this robot, it was possible to produce very rough maps of a simple environment constructed of cinderblocks. One such environment can be seen in Fig. 6. With a second robot, it was possible to map two environments simultaneously, as seen in Fig. 7.

5.2 Software Performance

The most significant problem with the mapping project is the presence of the WiFi link inside the control loop. Since the Julia mapping code is processing sensor data and producing motor commands on the host PC, it must send and receive data over WiFi at every control loop iteration. If the WiFi link is interrupted, then all control over the EV3 robot is lost.

In practice, running one robot at a time generally worked well, but adding a second robot resulted in frequent, temporary, losses of contact.

Figure 5: The robot exploring an environment made of cinderblocks.

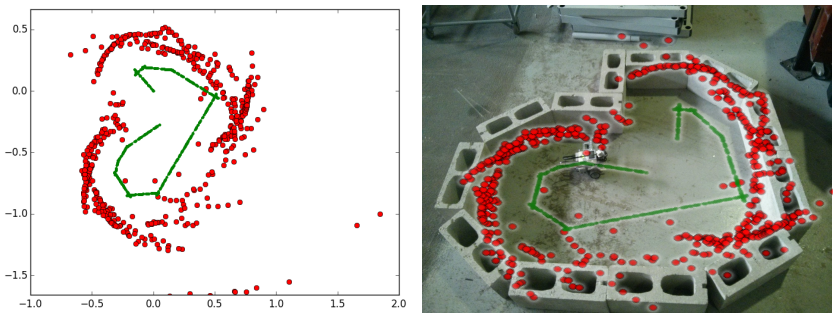


Figure 6: Demonstration of the `MappingRobots.jl` package. The robot spent 120 seconds traveling around an environment made of cinderblocks. Each red point represents a single ultrasound sensor return, plotted at its x-y position in the world using the robot's estimated pose. The green path represents the robot's estimated trajectory through the world. On the right, the points have been manually overlaid (and adjusted for perspective) on top of a photo of the environment.

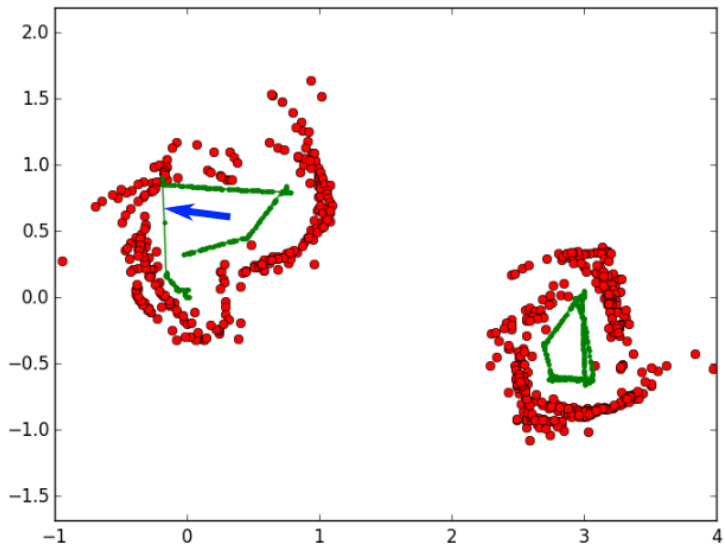


Figure 7: Demonstration of collaborative mapping. Two robots worked simultaneously to produce the pointcloud shown here. The red points are ultrasound sensor returns, and the green paths are the robots' trajectories. The section of the left path indicated by the arrow shows a temporary loss of communication between the host PC and the EV3 robot, during which no data about the robot's position was received.

Fortunately, the ZeroMQ layer was always able to re-establish connectivity without interrupting the task, but a total loss of motor control for several seconds is a substantial problem in a robotics application. The results of one such WiFi dropout can be seen in the very sparse data representing the left robot's estimated path in Fig. 7. In order to make more complex tasks possible, it will be critical to find a way to eliminate the WiFi link from the control loop, as will be discussed in Sect. 6.1.

6 Future Work

My most immediate future work will be to clean up and robustify the Julia code developed for this project so that others can use it. One important consideration is security of the Python server. As currently implemented, the server allows arbitrary files to be written and read on the EV3 remotely by anyone on the local network. A simple way to improve this will be to restrict the server to only access the virtual files corresponding to the motors and sensors, not any other (potentially sensitive) files on the device. This will still allow any user on the user's home network to remotely control the EV3, but not to damage the operating system.

6.1 Eliminating the WiFi Link

As discussed in Sect. 5.2, the fact that every command and piece of sensor data must travel over the WiFi link is a major limitation. There are a number of possibilities that we might explore to address this.

6.1.1 Move Julia onto the EV3

The most obvious way to remove the WiFi link is to remove or modify enough components of Julia to allow it to run directly on the EV3. With the Julia mapping code running onboard, the only communication over WiFi will be high-level, low-frequency commands, such as an instruction to start executing a mapping task.

6.1.2 Mount a Raspberry Pi Onboard

If a working Julia build cannot be created for the EV3, there may be another way to achieve similar performance. The Raspberry Pi is a series of small, low-cost computers which run standard Linux desktop operating systems. The Raspberry Pi Zero, for example, is sold for $5, but has 512 MB of RAM, compared to the 64 MB on the EV3, and a more modern ARMv6 architecture. There has been some success building working Julia versions for the Raspberry Pi, so it should be possible to run the mapping code on the Pi. The Pi could then be mounted directly on the robot, and could communicate with the EV3 over USB, eliminating the unreliable WiFi link.

6.1.3 Replace the EV3 entirely

Instead of adding a Raspberry Pi to the EV3 robot, it might also be possible to replace the EV3 brick entirely. The BrickPi is a custom device which allows Lego motors and sensors to be controlled directly by a Raspberry Pi over USB [10]. This involves some additional hardware, but might be a promising direction, since the Raspberry Pi is much more powerful than the EV3 computer.

6.1.4 Robot Operating System

Another possible direction to explore is the use of ROS, the Robot Operating System. ROS is a set of libraries for common robotics applications, including visualization, algorithms, and controls. One of the most basic elements of the ROS stack is the message-passing system, which allows different processes (potentially on different computers) to send and receive messages. It would certainly be possibly to replace the ZeroMQ layer with ROS messages over WiFi. This is unlikely to improve reliability, since the WiFi link would still be part of the control loop, but using a standardized library would make the code more likely to be compatible with other tools. There has been some very limited success getting ROS to run on the EV3, but as it is a very large library (hundreds of megabytes or more) and only fully supported on Ubuntu Linux, I chose not to explore it for this project.

7 Conclusion

This project demonstrated a complete solution for the use of Julia to control Lego Mindstorms robots, along with an application to demonstrate Julia's excellent parallel computing facilities. While the success was limited by the intermittent communication losses over the WiFi link between the host PC and the EV3, it was still possible to control two robots simultaneously from the same master process. In the future, I hope to eliminate the problematic WiFi link and create more robust robotics applications.

References

[1] P. Hintjens, "ZeroMQ: The Guide," 2015. [Online]. Available: http://zguide.zeromq.org/page:all

[2] The Lego Group, "History of Lego Robotics," 2015. [Online]. Available: http://www.lego.com/en-us/mindstorms/history

[3] "Not Quite C," 2007. [Online]. Available: http://bricxcc.sourceforge.net/nqc/

[4] "Next Byte Codes," 2011. [Online]. Available: http://bricxcc.sourceforge.net/nbc/

[5] "LegoMindstormsRobots.com," 2015. [Online]. Available: http://www.legomindstormsrobots.com/

[6] "Building Julia on ARM," 2015. [Online]. Available: https://github.com/JuliaLang/julia/blob/master/README.arm.md

[7] "Brickstrap," 2015. [Online]. Available: https://github.com/ev3dev/brickstrap

[8] R. Deits, "Building Julia on Lego Mindstorms EV3," 2015. [Online]. Available: https://groups.google.com/forum/#!topic/julia-dev/qaEjp5un9gE

[9] The Lego Group, "Lego Digital Designer," 2015. [Online]. Available: http://ldd.lego.com/en-us/

[10] Dexter Industries, "BrickPi," 2015. [Online]. Available: http://www.dexterindustries.com/brickpi/