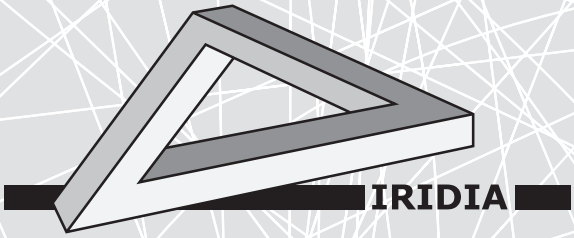



**Université Libre de Bruxelles**

*Institut de Recherches Interdisciplinaires*
*et de Développements en Intelligence Artificielle*

# ASEBA for ARGoS

Manuele Brambilla
Stéphane Magnenat
Carlo Pinciroli

**IRIDIA – Technical Report Series**

Technical Report No.
TR/IRIDIA/2009-016

May 2009

**IRIDIA – Technical Report Series**
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2009-016

Revision history:
TR/IRIDIA/2009-016.001 May 2009

# ASEBA for ARGoS

Manuele Brambilla mbrambilla@ulb.ac.be
Stéphane Magnenat stephane.magnenat@epfl.ch
Carlo Pinciroli cpinciro@ulb.ac.be
IRIDIA, Université Libre de Bruxelles, Brussels, Belgium

This document refers to:
revision 258 of ASEBA *studio*
revision 2687 of ARGoS

January 2009

# Contents

# Credits

ASEBA is the work of Stéphane Magnenat (http://people.epfl.ch/stephane.magnenat), Philippe Retornaz (http://people.epfl.ch/philippe.retornaz) and Francesco Mondada (http://people.epfl.ch/francesco.mondada) from EPFL (Ecole Polytechnique Fédérale de Lausanne - http://www.epfl.ch/).
The E-Puck project has been started at the Ecole Polytechnique Fédérale de Lausanne as a collaboration between the Autonomous Systems Lab (http://asl.epfl.ch/), the Swarm-Intelligent Systems group (http://swis.epfl.ch/) and the Laboratory of Intelligent System (http://lis.epfl.ch/).
The ARGoS simulator has been developed at IRIDIA (Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle - http://iridia.ulb.ac.be).
This manual is a modified version of the original ASEBA manual, written by Stéphane Magnenat.
Both the latest version of the original software and the manual can be found at http://mobots.epfl.ch/aseba.html.

# Chapter 1

# Introduction

A robot is a electromechanical machine that can interact with the world through sensors (e.g.: cameras and proximity sensors) and actuators (e.g.: wheels and arms). An autonomous robot is a particular kind of robot that can perform desired tasks without continuous human guidance.
But how is this possible? What is behind the behavior of an autonomous robot? It is the controller that gives a brain to a robot. It dictates the strategy of the robot and its reactions to external stimuli. A controller is a piece of software that, executed by the hardware of the robot, gives it the ability to accomplish tasks.
The goal of this manual is to give you a first look into the autonomous robot world. You will see how to write a robot controller using ASEBA [3, 4] and ARGoS [7, 6]. The first is an integrated development environment for mobile robot controllers, while the second is a simulator for robotic environment. At the end of this manual, you will be able to program a controller for a simple mobile robot, called E-Puck [1, 5], through ASEBA and test it in the ARGoS simulator. This manual is intended also to be a guide to ASEBA *studio* and to its scripting language.
The present guide includes information useful for beginner users up to experienced robot programmers who want to start learning ASEBA or need a reference for the scripting language.
The structure of this manual is the following: it starts with a panoramic over the E-Puck (Section 2) and its characteristics, followed by a small tutorial (Section 3) and an overview of ASEBA *studio* (Section 4). Finally there is a reference guide to the ASEBA scripting language with examples (Section 5).

# Chapter 2

# The E-Puck

The E-Puck is a mobile robot. Its target audience is high school and university students. The main goal of the E-Puck project is to develop a miniature mobile robot for educational purposes. To achieve this goal the robot has the following features:

- **Good structure** - The robot has a clean mechanical structure, simple to understand. The electronics, processor structure and software are a good example of a clean modern system.
- **Flexibility** - The robot can cover a large spectrum of educational activities and therefore has a large potential in its sensors, processing power and extensions. Potential educational fields are, for instance, mobile robotics, real-time programming, embedded systems, signal processing, image or sound feature extraction, human-machine interaction or collective systems.
- **User friendly** - The robot is small and easy to use on a table next to a computer. It needs minimal wiring and it has a long battery time.
- **Good robustness and simple maintenance** - The robot is able to resist to heavy use being simple, robust and cheap to repair.
- **Cheap** - The robot is cheap compared to other mobile robots.
- **Extensibility** - The E-Puck robot can be extended through a bus that includes most of the signals of the processor and some signals of the sensors and motors.
- **Open hardware** - All related documents and files (mechanical drawings, schematics, production files, source code) are distributed under an open hardware license.

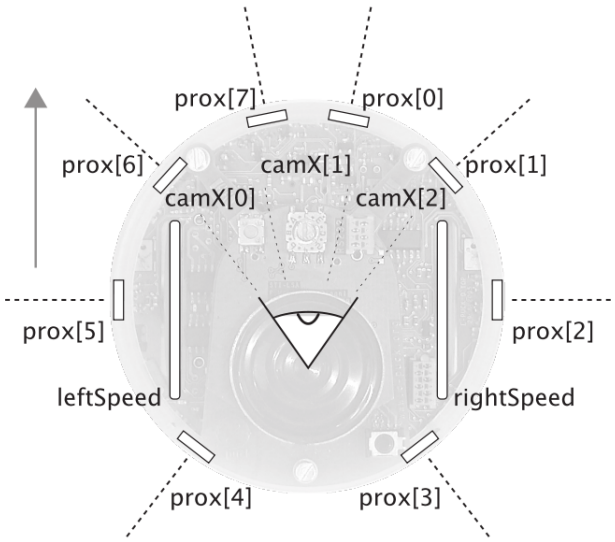



Figure 2.1: Some sensors and actuators of the E-Puck robot: the proximity sensors, the camera and the two wheels.

# Technical Characteristics [2]

| | |
|---|---|
| Diameter | 70 mm |
| Weight | 150 g |
| Battery | Li Ion removable 5Wh |
| Autonomy | 2-3 hours |
| Motors | two stepper motors 20 steps/rev |
| Gear type | included in motor block |
| Gear reduction | 50:1 |
| Resolution on wheels displ. | 0.1 mm |
| Mechanical structure | plastic body supporting PCBs, battery and motors |
| Surface of work | very flat |
| Processor | dsPIC 30F6014 @ 60MHz |
| processor type | 16 bit micro-controller with DSP core |
| RAM | 8k |
| FLASH | 144 k |
| Sensors | 8 IR proximity and light sensors<br>3D accelerometer<br>3 microphones<br>640 x 480 color camera<br>battery voltage |
| Actuators | 2 stepper motors<br>8 LEDs around the body, controlled independently in intensity<br>one speaker<br>one body light |
| Communication | RS232<br>Bluetooth<br>Infra-Red remote control |
| Programming env. | GCC integrated in Mplab |

Comments on the structure:

The E-Puck is based on a plastic body supporting the motors, the battery and the electronics. This plastic body is fully transparent allowing to see all elements of the robot.

The battery is placed on the bottom and can be easily extracted and recharged separately. A battery protection is implemented to avoid battery damage. Power on and low battery indicators help in understanding the status of the battery. The processor can read the battery level.

The two wheels are actuated by stepper motors with 20 steps/revolution through a reduction gear with a ratio of 50:1. The third contact point with the ground is made by the plastic body. The robot is designed to run on flat surfaces such as a table. Being well protected against dust, the robot should be able to run on a standard room ground.

The processor on board is a dsPIC 30F6014 running at 60 MHz (internal, corresponding to 15 MIPS). This processor has both a standard micro-controller structure and a DSP (digital signal processing) computation unit. Its 16 bits core is much more advanced than a PIC core (dsPIC has 16 registers and many DSP and C oriented instructions) and is designed to support C programming. The DSP core brings very high performances in signal processing applications.

The E-Puck is equipped with 8 infrared (IR) proximity sensors with a detection distance of some centimeters (3-4). Others sensors are a 3D accelerometer, 3 omni-directional microphones and a color camera with a resolution of 640x480 pixels. The processor is very well suited for the processing of the IR sensors, of the accelerometer data and the sound.

As actuators, in addition to the wheels, the E-Puck is equipped with a speaker, 8 red LEDs around the body and a green led inside the transparent body. On the speaker one can play any kind of sound. The 8 red LEDs and the green body led can be controlled in intensity.

The communication links supported by E-Puck are a standard RS232, an infrared remote control and Bluetooth. On Bluetooth there is a serial line emulation supported by any PC, making the communication and the development of PC software very simple.

The E-Puck is equipped with several extension connectors allowing to expand the system in several ways, with intelligent extensions or very simple interfaces.

# Chapter 3

# Tutorial

You will use two different programs that, working together, will let you program your robots and simulate their behavior.
ASEBA *studio* (developed by EPFL) is the controller development environment, where you will write your code, while ARGoS (developed by IRIDIA) is the simulator, where you will see the robots working according to your controllers. To start the simulator you must, first of all, open a terminal and execute this command in the simulator directory:

```
user@host:~/argos/$ bin/ahss \
                    -c xmls/epuck_aseba_base.xml
```

You should now see a window with the arena and the robots (Figure 3.1). Here are some useful commands you can use to interact with the simulator

- Ctrl-P : pause / unpause
- Ctrl-O : single simulation step
- Ctrl-T : toggle textures
- Ctrl-S : toggle shadows
- Ctrl-X : exit

You can change the camera position by clicking + dragging in the window:

- Left button - pan and tilt.
- Right button - forward and sideways.

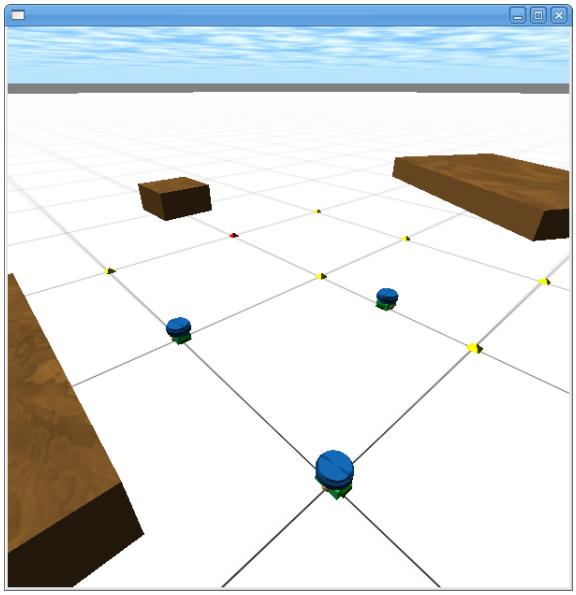


Figure 3.1: Screenshot of ARGoS.

- Left + Right button (or middle button) - sideways and up.

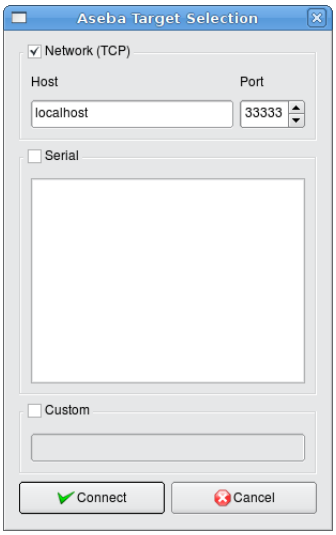
Once the simulator is running, you have to start ASEBA *studio* to program your robot. This could be done by opening a terminal window and executing the following command:

```
user@host:~/argos$ bin/asebastudio
```

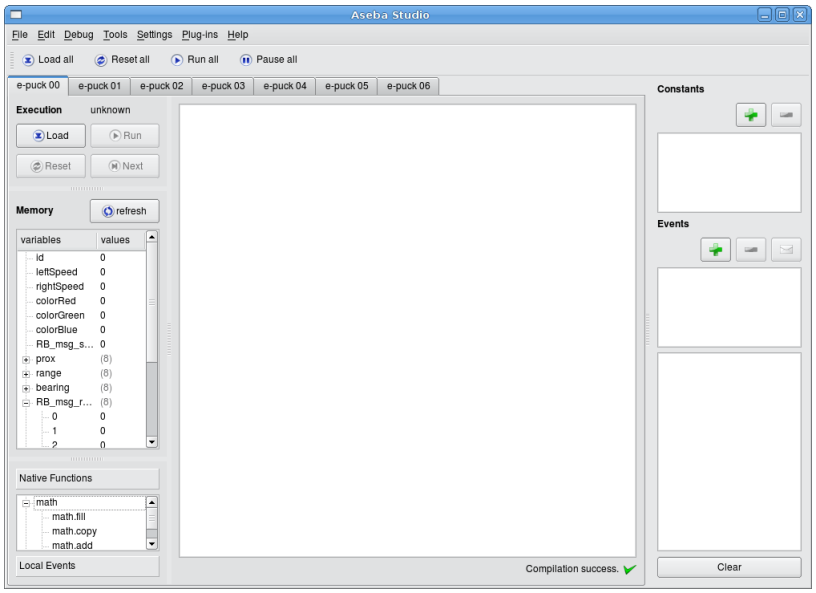
Then you have to connect ASEBA *studio* to the simulator: when the dialog (Figure 3.2(a)) shows up just leave the default port and click on "connect"; you should now see the *studio* (Figure 3.2(b)). A detailed description of *studio* is given in Section 4.

# Your first robot controller

Now it is time to write your first controller!
To program a robot, you first have to understand how it works. A robot interacts with the world in a loop: it perceives the state of the world through its sensors, takes some decisions with its onboard computer, and does some actions with its actuators; those actions change the state of the world, and




(a) *studio* connection dialog.




(b) *studio*.

Figure 3.2: ASEBA *studio*.

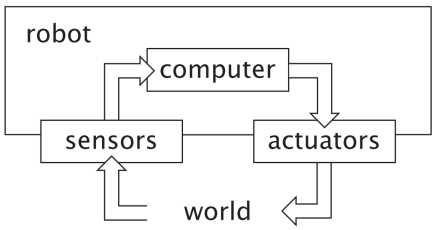



Figure 3.3: A robot interacts with the world in a loop through its sensors and actuators.

the robot perceive this new state when it reads its sensors again (Figure 3.3). With ASEBA *studio* and the simulator, you can program a simulated E-Puck robot. The simulator is built to be as accurate as possible with respect to the real robot (the characteristic of the real robot are listed in Section 2): in this way you can write your controller once, test it in the simulation and port it to the real robot.
Now that you know how a robot interacts with the world, let us write your first robot controller. A complete high level description of ASEBA scripting language, with examples, is available in Section 5.
In the text editor of the studio window, write the following lines:

```
leftSpeed = 100
rightSpeed = -100
```

Click "Load" and then "Run". You should see your robot turning on the spot. This code sets the speed of the wheels when the robot starts. Since these are the only instructions in the script, the robot keeps turning until the simulator is stopped.
In order to have a more complex behaviour, we should allow the robot to perceive its environment, for instance the obstacles, and make decisions, for example to stop.

# A more elaborate robot controller

To interact with the world continuously, the robot must execute some instructions periodically. This is achieved with the `onevent timer` keyword. For example, using a front proximity sensors, we can sense if there is an object in front of the robot and stop:

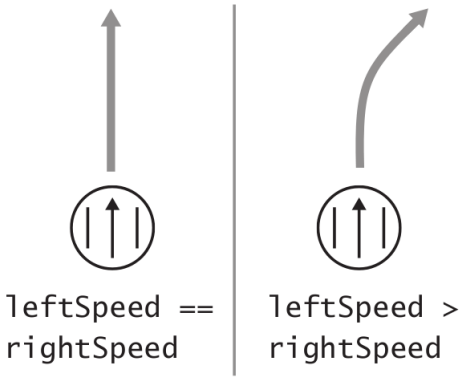



Figure 3.4: The robot trajectory changes with respect to the speed difference between the wheels.

```
# initial setup - go straight
   leftSpeed = 100
   rightSpeed = 100

# sense the world and
# take a decision at every step
   onevent timer
   if prox[0] > 100 then # obstacle
      leftSpeed = 0
      rightSpeed = 0
   end
```

The proximity sensor gives a perception of the distance between the robot and an object. The output of the sensor is a numeric value between 0, if there are no objects in sight, and 3000, the robot is next to an obstacle. We have just written a controller that stops the robot as soon as the proximity sensor senses an obstacle.
Of course stopping when sensing an obstacle is good but we would like to be able to do something else, for example turning. To have the robot wander around, we have to understand how it moves.

The E-Puck is an example of a differential wheeled robot; this type of robot changes its direction by setting different speeds to its right and left

wheels (Figure 3.4). If each wheel has the same speed, the robot goes forward; otherwise it turns; if the speeds are opposite, the robot turns on the spot. Industrial caterpillar vehicules use the same movement modality.
Now we are able to write a controller that makes a robot that turn softly when it senses a far obstacle, turn on itself when next to an object, and goes straight in any other occasion. We take input from the two frontal proximity sensor to be safer. Here's the code for obstacle avoidance:

```
leftSpeed = 100
rightSpeed = 100

onevent timer
if prox[0] > 1500 or prox[7] > 1500 then
   leftSpeed = -100
   rightSpeed = 100
else
   if  prox[0] > 500 or prox[1] > 500 then
      leftSpeed = 30
      rightSpeed = 100
   else
      leftSpeed = 100
      rightSpeed = 100
   end
end
```

It could be a good idea to see visually when a robot senses an obstacle. How can we do it? The E-Puck is equipped with 8 LEDs that are a really good way to show an internal status. So we just need to turn on the LEDs when sensing an object. This, as all the commands to actuators, can be done simply by setting the value of the LEDs to 1. To do that we could use a *for* cycle to set all the LEDs variables to 1, but *"Leds"* is an array and the best way to change all the values of an array is to use the *fill* function, that has for parameters the target array and the number to copy. Here's is the modified code:

```
...
   if  prox[0] > 500 or prox[1] > 500 then
      leftSpeed = 30
      rightSpeed = 100
```

```
        call math.fill(Leds, 1) #turning on
    else
        leftSpeed = 100
        rightSpeed = 100
        call math.fill(Leds, 0) #turning off
    end
...
```

# Your turn

Now it is your turn! Feel free to explore, the robot is in a simulator and does not risk any harm. Try to set the velocity as a constant: you can do this by pressing the green "plus" button in the top right part of ASEBA *studio*, just under *Constants*, give it a name, for instance *MAX_SPEED*, set the value by double-clicking the number next to the constant (now it should be 0) and finally use the constant in your code as it was a variable. In this way you can modify the value of this constant easily and for all the robots (if you have more than one). Another thing to try is to command an actuator on-the-fly: in the left part of ASEBA *studio*, under *Memory* there is the list of all the variables, sensors and actuators; if you double click on any of the actuators or variable values (sensors are readonly) you can set the number you want and see how the robot reacts to this!
Try to experiment with the simulator and ASEBA *studio*, find all the possibilities of the E-Puck and get a good understanding of the sensors, actuators and dynamics of the robot.

# Chapter 4

# ASEBA *studio*

ASEBA *studio* is an integrated development editor, created by EPFL, within which we edit and debug the scripts. It provides the following features:

- **Concurrent editing** - Each robot has its own tab with its script, memory content, execution situation, sensor and actuator status, and debugging commands. Every tab provides the command to load, reset, start/pause, and execute step-by-step a single controller. In addition, a toolbar provides the same commands to affect all the E-Pucks. This allows us both an overall control of the group and a specific control (very useful during the debug phase) of each script.

- **Syntax highlighting** - The script editor highlights the syntax and colors errors in red. This increases the readability of the scripts and helps pointing out mistakes.

- **Instant compilation** - *studio* recompiles the script while the developer is typing it. The result of compilation (success or a description of the error) is displayed below the editor. This permits the correction of errors as soon as they appear, which increases the quality of the code. By clicking on the error displayed *studio* automatically scroll to the wrong part helping the user to correct it.

- **Variables inspection** - A list of the variables available on each robot along with their values is available in each tab. We can update this list with a single click on the *Refresh* button. This list provides a quick overview of the state of the E-Pucks. This also allows you to modify on-the-fly the values of actuators and variables, you can do that just by double-clicking on the variable you want to change.

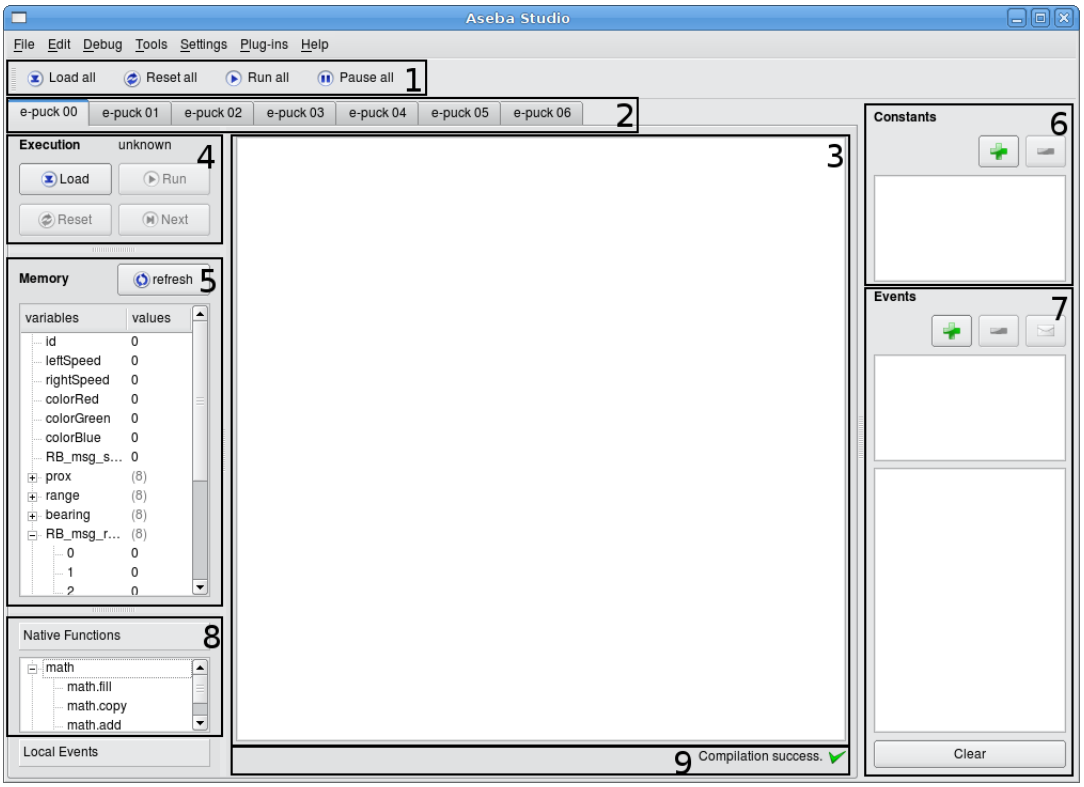



Figure 4.1: Screenshot of ASEBA *studio*. The enclosed elements are the following: 1. General commands toolbar - 2. Unique tabs for each robot - 3. Editor with syntex highlighting - 4. Robot specific debug commands - 5. Content of memory with the names and values of variables - 6. Constants definition - 7. Events definition and log - 8. Native functions documentation - 9. Compilation output and errors.

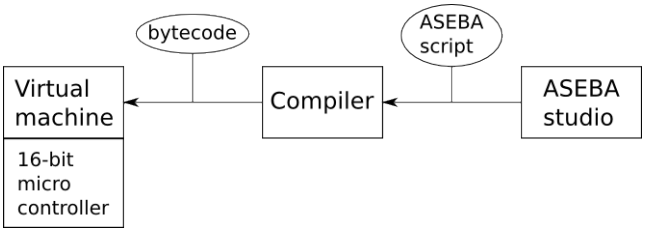



Figure 4.2: The chain of operations from the script to the robot.

- **Debugger** - *studio* integrates a debugger; for each robot, it shows the current execution status. It supports continuous execution, step by step, and breakpoints. A right click inside the script editor allows us to set or clear a breakpoint on a specific line. After a breakpoint or a step, the values of the variables are updated.

- **Events** - We can specify the names of the events, and by double-clicking on a name, we can send the corresponding event. Below the list of events names, a log shows the recent events along with their time stamp and parameters. Note that with E-Pucks events can only be used as debugging "flags", i.e. showing the programmer that a certain event has been emitted. For the syntax see Section 5.

- **Native functions** - *studio* lists the native functions available on each E-Puck. These are useful mathematical functions that can be used during the developing of the software. It is strongly suggested to use these functions instead of re-writing your own for efficency and space optimization. The tooltip of each function gives a short documentation about the function.

- **Constants definition** - We can define constants that are available for all the robots. This provides a way to control common parameters among all robots. Try to use a constant everytime you need a common numerical input. In this way it will be easier for you to change the parameters if you need and for others to understand what you write. Constants definition and using is explained in Section 5.

The chain of operations behind ASEBA is this (figure 4.2): first you write your controller in ASEBA *studio* using the scripting language, then the internal compiler translates the script into bytecode (a set of instruction) that is used by the virtual machine inside the robot micro-processor.

# Chapter 5

# Scripting Language

Using ASEBA *studio*, you program the robots by writing scripts in a simple language. Syntactically, this language resembles Matlab (and Java/C); semantically, it is a simple imperative programming language with a single basic type (16 bits signed integers) and arrays. Here are its features:

- **Comments** - every time you need to comment or write something that doesn't need to be compiled use '#'. Comments begin with a '#' and terminate at the end of line.

```
# this is a comment
var b # another comment
```

- **Variables** - Variables refer either to single scalar values or to arrays of scalar values. The values are only integers ranging from -32767 to 32767, which is the range of 16 bits signed integers. Remember these constraints when operating with numbers because if the result is out of these bounds an integer overflow happens (e.g. 32767 + 1 = -32768). Another thing to bear in mind is that division results are rounded to the smallest integer (e.g. 999/1000 = 0).
  You can access arrays elements using the usual square parenthesis operator; arrays indexes begin at zero.
  All variables must be declared at the beginning of the script.

```
# simple declaration
var a
# declaration and initialization
var b = 0
# array declaration
var c[10]
# array declaration and initialization
var d[3] = 2, 3, 4
```

- **Expressions and assignations** - Expressions allow mathematical computations and are written using the standard mathematical syntax. Assignation uses the symbol = and sets the result of the computation of an expression to a scalar variable or to an array element. ASEBA provides the operators +, -, *, /, % (modulo), << (left shift), and >> (right shift). Operator precedence is *, /, %; followed by + and -; followed by << and >>. To evaluate an expression in a different order, we can use a pair of parenthesis to group a sub-expression.

```
a = 1 + 1
b = (a - 7) % 5
b = b + d[0]
c[i] = d[i]
```

- **Conditionals** - ASEBA provides two types of conditionals: `if` and `when`. ASEBA provides the operators ==, !=, >, >=, <, and <=. Both `if` and `when` execute a different block of code whether a condition is true or false; but `when` executes the block corresponding to true only if the last evaluation of the condition was false and the current is true while `if` does so each time that the condition is true, and else each time it is false. This allows the execution of code only when something changes, which is convenient for robotics applications. The syntax is visible in the example.

```
if a - b > c[0] then
   c[0] = a
else
   b = 0
end
```

```
when a <= b do
   c = d * 37
end
```

- **Loops** - Two constructs allow the creation of loops: `while` and `for`. A `while` loop repeatedly executes a block of code as long as the condition is true. The condition is of the same form as the one `if` uses. A `for` loop allows a variable to iterate over a range of integers, with an optional step size. The syntax is shown in the example.

```
while i < 10 do
   v = v + i * i
   i = i + 1
end
```

```
for i in 0:8 do
   t[i] = a[i]*2
end
```

```
for i in 1:10 step 2 do
   v = v + i * i
end
```

- **Native functions** - ASEBA script is designed to be simple to allow a quick understanding even by novice developers and to implement the

virtual machine efficiently. To implement complex or resource consuming processing, we provide native functions that are implemented in native code for efficient execution. For instance, a native function is the natural way to implement a scalar product. Native functions are safe, in that they specify and check the sizes of their arguments, which can be constants, variables, or array accesses. In the latters, we can access the whole array, a single element, or a sub-range of the array. Native functions take their arguments as reference and can modify their contents but do not return any value. Native functions are called by the `call` keyword.

```
var a[3] = 1, 2, 3
var b[3] = 2, 3, 4
var c[5] = 5. 10. 15
var d
call vecprod(a, b, d, 3)
call vecprod(a, c[0:2], d, 3)
```

- **Events** - Events in ASEBA are mostly used as debug tools. A programmer can use them as a non-blocking signal of a special condition. A script can send an event by using the `emit` keyword. Events must be defined in the "event" tab. Flags raised are shown in the same tab. An event can also have a parameter that is showed when the corrispondent event is called.
  There is a very special native event called `timer`. The `timer` event is generated every time-step and it is used by the programmer to make the robot react to external stimuli. All the code following the `onevent timer` keyword is executed at every time step and usually is the core of the controller. Normally the code is divided in three parts: variable declaration, variable inizialization and the code that must be executed every time-step. This is the piece of the controller that analyzes new sensor data and operate the actuators.

```
if prox[0] > 500 then
    emit collision_event
end
```

```
onevent timer
if prox[0] > 500 then
   leftSpeed = MAXSPEED
   rightSpeed = -MAXSPEED
else
   leftSpeed = MAXSPEED
   rightSpeed = MAXSPEED
end
```

# Bibliography


[1] Michael Bonani. E-puck. `http://www.e-puck.org`.

[2] Michael Bonani, Philippe Retornaz, and Francesco Mondada. E-puck flyer. `http://iridia.ulb.ac.be/wiki/images/2/2c/E-puck-flyer-V1.pdf`.

[3] Stéphane Magnenat, Philippe Retornaz, and Francesco Mondada. Aseba. `http://mobots.epfl.ch/aseba.html`.

[4] Stéphane Magnenat, Philippe Rétornaz, Basilio Noris, and Francesco Mondada. Scripting the swarm: event-based control of microcontroller-based robots. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR) 2008*, 2008. Workshop Proceedings ISBN: 978-88-95872-01-8.

[5] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a Robot Designed for Education in Engineering. In *9th Conference on Autonomous Robot Systems and Competitions*, volume 1, pages 59–65, Portugal, 2009. IPCB – Instituto Politécnico de Castelo Branco.

[6] C. Pinciroli. Object retrieval by a swarm of ground based robots driven by aerial robots. Diplôme d'Etudes Approfondies en Sciences Appliquées thesis, IRIDIA, Université Libre de Bruxelles, 2006.

[7] Carlo Pinciroli, Marco Dorigo, and Mauro Birattari. Argos. `http://www.swarmanoid.org/swarmanoid_simulation.php`.