THE BEHAVIOR LANGUAGE FOR PC: USERS GUIDE

INDEX

1 Introduction 2
2 The Behavior Language 4
2.1 Program structure 4
2.2 Definitions 5
2.2.1 Constants 5
2.2.2 Types 5
2.2.3 Variables 6
2.2.4 Functions 7
2.3 Expressions 8
2.4 Modules 10
2.5 Behaviors 11
2.6 Rule syntax
Whenever 12
2.7 Instructions 14
2.8 Macros 16
2.9 Debugging 16
2.10 Comments 17
3 User Interface 18
3.1 The Compiler 18
3.2 Examples 19
4 References 21
Appendix A
The C program 22
Appendix B
BL Warnings and Errors 25
Appendix C
The BL grammar 28
Appendix D
Reserved words 34
Appendix E
Genghis Interface 36

THE BEHAVIOR LANGUAGE FOR PC: USERS GUIDE[1]

Josep Mª Porta
Institut de Robòtica i Informàtica Industrial (IRII)
UPC-CSIC
Autumn, 1996

Abstract

This document is a description of a new version of Brooks's Behavior Language (a language for programming autonomous robots). First we explain the reasons that forced us to develop a new compiler. Then we explain how to write a correct program for the new compiler. As you will see, many new programming constructions change or are new compared with the existing Behavior Language (modules, type declaration, debugging utilities,...) but the original philosophy is preserved (programs are groups of behaviors containing reactive rules running in parallel that share information through explicit message passing). To conclude the document, we explain how to use the new compiler to create an executable file.

[1]This work has been partially supported by a grant of the "Comissionat per a Universitats i Recerca de la Generalitat de Catalunya".

1 Introduction

The Behavior Language (BL) is a programming language for autonomous robots designed by Brooks [1] following the principles of the subsumption architecture [2]. We have use this programming language to successfully implement some controllers for a Genghis II robot.

The steps to execute a Behavior Language program are:

- First we have to write the program in Behavior Language.
- Then we compile it using a compiler embedded in MACINTOSH COMMON LISP. This compiler generates a program in the robot's microprocessor machine language (a Motorola 68HC11 in the case of our robot).
- And lastly we transfer this program to the robot trough serial line.
- Now we can let the robot run its new program.

This process has many advantages:

- The robot works in a completely autonomous way.
- The behavior of the robot can be highly reactive since changes in the sensor readings can be processed as soon as they happen.

…

But it also has some disadvantages:

- The available memory for program and data storage in the robot is quite short (32 KBytes in our case).
- The robot's microprocessor is quite limited (it cannot perform operations with floating point numbers,...).
- Long program debugging is slow because of the inefficiency of a compiler working over LISP and because of the serial line low transference rates (even when there are not transmission errors!!). This makes that, for example, we expend more than 2 minutes to be ready to test (that includes compile plus transfer) every little change of a 27 KB program. Since such a long Behavior Language program usually contains some tens of parameters to be carefully adjusted, the complete tuning of the controller can take some weeks.
- Programs are difficult to monitor. They are executed by the robot and often not enough information can be extracted at run time (using the serial line to display it on a terminal).
- There is a lack of standard software (floating point emulators,...) to be included in the user programs. Every piece of software has to be programmed from scratch.

Our work with Genghis has been extremely limited by these problems. We could solve some of them augmenting the robot capabilities: increase the memory, improve the serial connection,... But all those solutions are not available in our case since we can not easily modify our robot (it wasn't designed to be extended). So, we thought in another solution. The idea is to implement our own version of the Behavior Language compiler to generate programs executables by an IBM compatible personal computer running DOS linked to the robot by the serial line. This serial line is used to transmit sensors' reading from the robot to the computer and actuators' commands from the computer to the robot. The PC executes the

main process while the robot plays a role of a peripherycal device. This allows us to use all the PC resources (memory, computational power,...). In order to include already existing software in our programs we choose to use C as the intermediate code of our compiler. Figure 1 shows the general structure of the compiler.

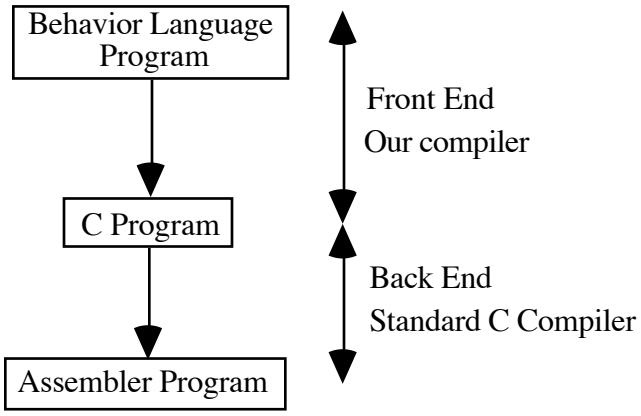


Figure 1: Compiler Structure

The Back End can link our Behavior Language program with any existing C program (a robot simulator,...).

But all these advantages are not for free. The problems of this solution are obvious: we are losing the possibility of operating the robot autonomously (it must be always connected to the computer) and the reactivity of our programs decreases because of the communication delays introduced by the serial line.

In the following sections we describe what components we can use in a behavior language program (that are not exactly the same than those described by Brooks in [1]) and how we can use the compiler.

2 The Behavior Language

2.1 Program structure

A program written in BL is organised in a hierarchy of modules as shown in Figure 2:

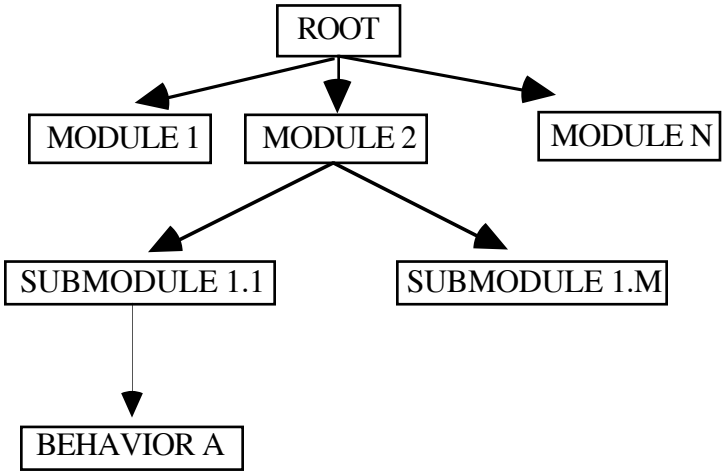


Figure 2: Module Hierarchy

The leaves of the tree are behaviors, the basic programming unit of the behavior language. A behavior is a set of reactive rules that controls one specific aspect of the robot conduct. Every one of these reactive rules is executed in parallel with the rest of the behavior's rules and with other behaviors' rules.

A behavior has access to all definitions (constants, variables, types and functions) declared in any module included in the way from that behavior to the root of the hierarchy. The root of the tree contents the definitions available from everywhere (for example: functions to read the robot's sensors,...) while each behavior contents the most private (local) definitions.

Behavior can share information through connections (pipes from one behavior to another one) conforming a net of interrelated processes.

The clause "include" allows the programmer to split the program in various files. It has the following syntax:

include "file-name**"**

and it can be used recursively. This facilitates the process called "layering" in which new capabilities are achieved by adding new software layers to existing programs. This process allows an incremental construction and testing of robot controllers.

Finally, standard C libraries can be used in a Behavior Language program using the clause "lib":

lib "standard-library**"**

C functions are used as normal BL functions but BL compiler ignores them (C compiler will consider them). Once a lib clause appears in a program, the use of undefined functions is not longer an error but turns into a warning.

2.2 Definitions

2.2.1 Constants

In our BL there are four types of constants:

- Integers: In decimal or in hexadecimal format. For example:
  1, 01, 10 or 1234 are decimal integer constants
  #x0A0F or #x12E are hexadecimal integer constants
- Floating point numbers: For example:
  0.33, 1.0, 1e-5 or 0.35E9 are floating point constants
- Characters: For example:
  'a', '.','$' or '5' are character constants
- Strings: Any text inside "" is considered a string constant. For example:
  "hello world" is a string constant

If we want to define a constant with a specific name we can use the clause "constant":

constant name **=** constant expression **;**

For example:

constant True=1;

constant False=0;

constant neg=-1;

constant Pi=3.1415;

constant surface=2*Pi*9;

2.2.2 Types

The pre-defined types in our version of the BL are:

- Integer: That can be signed or unsigned (if nothing is indicated unsigned is supposed)
- Char: We can also specify the sign (unsigned is again the preferred option)
- Double

They are equivalent to the C types with the same name.

Our BL includes two ways to construct new types:

- Array: To create fixed size sets of elements of the same type. Its syntax is:

  Array [size**] of** element type

  Like in C, the elements of a size 'n' array are numbered from 0 to n-1.
- Record: To join different type elements in a single structure. Its syntax is:

  Record element **EndRecord**

  And each element is defined as follows:

  element name **:** element type **;**

The clause “type” allows us to assign a name to a new type:

type type name **:** type definition **;**

Examples of type definition are:

```
type increment : signed char;
type uchar : unsigned char;
type string10 : array [10] of char;
type employee : record
                    name : string10;
                    salary : unsigned integer;
                    salary_change: increment;
                    height : double;
                    endrecord;
type corporation : array [100] of employee;
```

We can also declare pointers to pre-defined type or to already defined types. For instance:

```
type integer_pointer : pointer to integer;
type char_pointer : uchar;
type pp : pointer to pointer to unsigned char;
```

2.2.3 Variables

The BL variables are defined with the following syntax:

var var name [**:** type] [**=** initial value]**;**

Where the items in brackets are optional. If no type is specified then the variable is assumed to be a “signed char” variable. This is because most the functions for reading the robot’s sensors uses this type as a return type. The initial value of the variable is set at random unless a special value is state. The initial value and the variable should be of the same type.

There is a special type that can be used in the variable definition: the monostable type. A monostable describes a Boolean variable that initiay is false, once triggered, holds true for a fixed time and then returns to false. The way to declare a monostable variable is:

var monostable name **: monostable (** period **);**

There is a special function to trigger a monostable variable:

trigger (monostable name **);**

that can be used as a normal function.

There are two global pre-defined variables:

- CLOCK: Contents the number of ticks since the program execution has started. With our robot, every tick corresponds to 0.02 seconds (with other robots this time could change and this will change all the pre-defined times in the compiler). The type of this variable is unsigned integer.

- STOP_GENGHIS: When it is different from 0 the program stops. The type of this variable is unsigned char.

Variables can appear in any declaration area of any module or behavior. If we use them in a module then we are setting up a shared structure across that module's behaviors and this is strictly against Brooks' subsumption architecture philosophy. In this architecture the only way to share information between behaviors is the explicit message passing (also included in our BL version) so we have to be carefull with these constructions.

Examples of variable definition are:

```
var leg0pos;
var force:double=3.0;
var pi:pointer to integer;
var name:array[10] of char;
var complex:record a:double; b:double; endrecord;
var person:employee; //From the last section examples
var frequency:array[27] of record letter:char; times:integer;
endrecord={{'a',0},{'b',0}};
//Note that there is no need to initialise the whole array
```

2.2.4 Functions

BL allows us to define functions with the following syntax:

```
Function function name ( parameters ) [: return type]
    Local variables definition
Begin
    Code
    Return (expression);
EndFunction
```

The format to specify parameters is:

parameter name **:** parameter type

We must do some observations about function definition:

- The local variables are defined as normal variables (see the previous section).
- The type of the returned expression should be equal to the return type of the function.
- The return clause can appear anywhere in the code part of the function.
- The return type is optional. If we not specify any return type then we are defining and action (a function with no return). In this case it is not necessary to include the return clause in the function body.
- BL functions can not be recursive.

For example, a function to calculate the factorial of a small positive integer could be defined as follows:

```
Function factorial (n:unsigned char):unsigned int
    var i:unsigned char;
```

```
        var r:unsigned int=1;
Begin
        For i=2 to n do
                r=r*i;
        EndFor;
        Return(r);
EndFunction
```

The BL compiler is not case sensitive except for the name of the variables, constants and functions. This is because our BL compiler translates these constructions directly into C structures and it's well known that C compilers are case sensitive.

There is a special kind of functions in our version of BL: the initialising functions. Those are functions that are executed before any rule become active and can be used, for example, to initialise variables, establish parameters,...

The syntax to declare an initialising function is:

Initfunction
        Code
EndInit

2.3 Expressions

In previous sections we have spoken about expressions. Now is the moment to define them more formally. We will consider that:

- Any constant or variable is an expression and the type of the expression is the same as the type of the constant or the variable.
- If e is an expression of type t and op is a valid unary operation over elements of type then

  op e

  is an expression and its type is the resulting type of op.
- If e1 and e2 are expressions of type t1 and t2 respectively and op is a valid binary operation between elements of type t1 and type t2 then

  e1 op e2

  is an expression and its type is the resulting type of op.
- If e is an expression then (e) is also an expression. The type of this new expression is the same as the type of e.
- If f is a function with n ($n \geq 0$) parameters and $e_1,...,e_n$ are expressions of the correct type then

  $f(e_1,...,e_n)$

  is an expression and its type is the return type or the function.
- If e is an expression and t is a type name then

  <t>e

  is an expression and is type is t. t should be a pre-defined type a user defined type or a pointer to one of the previous.
- If e is an expression of a record type t and n is a name of a field of t then

  e.n

is as expression and its type is the type of the field n of t.

- If e is an expression of type t, t is a pointer to a structure and n is a name of a field of that structure then:

  e->n

  is as expression and its type is the type of the field n.
- If e is an expression of type t, t is an array and e2 is an integer expression then

  e1[e2]

  is an expression and its type is the type of the elements of t.
- Nothing else is an expression.

To conclude this definition we have to specify what are the valid unary and binary operations in BL. The arithmetic operations are:

- Unary sign specification: +, -
- Unary self-increment (or decrement): ++, -- (as a pre or post operator)
- Product, division and remainder: *, /, %
- Addition and subtraction: +, -

The logical operations are:

- NOT
- Comparison: >, <, >=, <=
- Equal and different: ==, <>
- Conjunction: AND
- Disjunction: OR

Any expression can be considered a logical expression since, like in C, BL takes as True any value different than 0.

The unary addressing functions:

- Dereferencing: If X is a pointer to ‘t’ expression then “*X” is a type ‘t’ expression.
- Referencing: If X is a variable then “&X“ returns the memory address of the variable X.

The bit level operations are:

- Left to right and right to left shifts: >>, <<
- And: &
- Xor: ^
- Or: |

The assignment operations are:

- Simple assignment: =
- Product assignment: *=
- Quotient assignment: /=
- Remainder assignment: %=

- Sum assignment: +=
- Difference assignment: -=
- Bitwise AND assignment: &=
- Bitwise OR assignment: |=
- Bitwise XOR assignment: ^=
- Left shift assignment: <<=
- Right shift assignment: >>=

The next table presents the precedence relation between operations:

Higher precedence
Function calling: () Array subscript: [] Field operators: .,->
+ ,- (Unary), ++, --, &, *, NOT
*,/,%
+,-
>>,<<
>=,<=,>,<
==,<>
&
^
|
AND
OR
=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=
Lower precedence

2.4 Modules

Once we start a program in BL we can consider that we are writing the Root module (see figure 2). If we want to declare a new sub-module we can use the construction:

Module module name

Module contents
EndModule

The module can content definitions (constants, types, variables and functions), sub-modules (declared using the same construction) and behaviors.

There is a special clause to control the module execution:

Activation activation expression **;**

This construction can appear anywhere in the module (no inside sub-modules or behaviors because then it would be considered that particular sub-module or behavior activation rule). If the activation expression is false (evaluates to zero) then all the behaviors of that module (or of sub-modules of that module) stop.

2.5 Behaviors

Behaviors are the basic unit of a BL program. A behavior controls a specific aspect of the robot acts. The syntax to declare a behavior is:

Behavior behavior name
inputs declaration
outputs declaration
local definitions
Processes
Reactive Rules
EndBehavior

The behavior name should be unic in all the program.

The input/output declaration allows to create pipes between behaviors. Inputs are declared as follows:

Inputs input declaration$_1$**,**.....**,**input declaration$_n$

An each input declaration is:

[**additive**] input name [**:** input type name] [**=** initial value] [**from** (behavior$_1$**,**output$_1$[**priority** priority$_1$])**,** …., (behavior$_n$**,**output$_n$ [**priority** priority$_n$])]

The input type name should be a pre-defined type (integer, char or double), a user defined type using the **type** clause (see section 2.2.3) or a pointer to a type. If no special type is declared then the input is supposed to be "signed char".

The type of the initial value (if specified) must be equal to the type of the input.

The clause **from** indicates witch outputs of witch behaviors are linked to the presently declared input.

The priority of a connection is used to break the tie when two output ports are sending messages to the same input port: the messages coming from the most priority connection defines the new value of the input port and the rest of messages are forgotten

If an input is **additive** then all messages to this input are accumulated. Only double, char and integer inputs can be additive.

There is a special function that operates over input registers:

Received(input name **)**

This functions returns true when there is a new data in the input register. From the point of view of each reactive rule a register contents new data if someone has changed its value after the last execution of the rule.

The syntax to declare outputs of the present behavior is:

Outputs output declaration$_1$**,**.....**,**output declaration$_n$

Each output declaration is:

output name [**:** output type name] [**to** (behavior$_1$**,**output$_1$ [**priority** priority$_1$])**,** …., (behavior$_n$**,**output$_n$ [**priority** priority$_n$])]

The clause '**to**' indicates witch inputs will receive the messages that the present behavior sends through that output.

Again, the output type name should be a pre-defined type or a user defined type. If there is a connection between two ports then the origin and destination ports must be of the same type, otherwise the compiler will produce an error.

There are two special functions to manage inputs and outputs:

- **Output** (output port**,** expression [**,** period expression])

  It sends the result of evaluating the expression to all inputs ports connected with the mentioned output port. The period specifies the amount of time that the present message PREVALECE over less priority new incoming messages. If no period is indicated 4 ticks (0.08 seconds with Genghis) is chosen. This is twice the execution period of a usual reactive rule. That allows a high priority process to take control over a specific input port .

- **Inhibit** (behavior name**,** port name [**,** period expression])

  It interrupts any message coming from (if it is an output) or to (if it is an input) a port for a fixed time. If no period is indicated 4 ticks (0.08 seconds) is the pre-defined option.

There is an explicit way to connect two behaviors:

Connect (behavior **,** output_port) **to** (behavior$_1$ **,** input_port$_1$)**,** (behavior$_2$ **,** input_port$_2$) **,...,** (behavior$_n$ **,** input_port$_n$)

The local declaration of the behavior can include:

- Constants declarations.
- Type declarations.
- Variable declarations.
- Functions declarations.
- An activation rule (with the same syntax as the module activation rules).

The behavior's processes are a collection of reactive rules as described in the next section.

2.6 Rule syntax: Whenever

The BL rules specify what to execute and when to do it with the following syntax:

Whenever boolean expression [**with time** expression] **do**
    code
EndWhenever;

Every one of these rules can be considered as an independent process working continuously in parallel with the other rules of the program.

The semantic of this programming construction is:

```
Every X seconds:
        If E is true execute C
```

X (called the period of the rule) can be defined thanks to the "with time" clause, E (the condition) is the boolean expression after the "whenever" clause and C is the code inside the whenever. The pre-defined period of all rules is 2 ticks (0.04 seconds if we are working with Genghis).

For example:

```
Whenever 1 do
        a=a+1;
EndWhenever;
```

Increments the variable 'a' every 0.04 seconds and

```
Whenever (a>3) or (a=1) with time 0.5 do
        a=0;
EndWhenever;
```

Checks every 0.5 seconds if 'a' is greater than 3 or equal to 1 and in this case, sets 'a' to zero.

There are two really useful constructions to use in a whenever condition (although they can be used everywhere in the program):

- Monostables: For example:

  ```
  Whenever Not (timer) do
          If a>4 Then
                  trigger(timer);
                  a=0;
          EndIf;
  Endwhenever;
  ```

  When 'a' is greater than 4 its value is set to 0 and the rule turns itself off for the period of the monostable 'timer' (this may allow other rules to change 'a'). Observe that this whenever is equivalent to:

  ```
  Whenever Not (timer) and a>4 do
          trigger(timer);
          a=0;
  Endwhenever
  ```

- Received: To awake a rule as soon as new data arrives into an input port. For example:

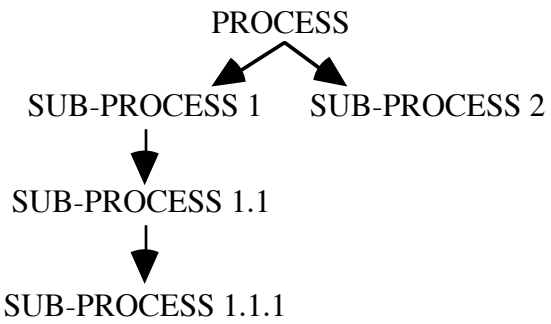
  ```
  Whenever received(data) do
          printf("%d ",data);
  EndWhenever;
  ```

  If there is a received clause in a whenever condition and no rule period is defined then the pre-defined period is 1 tick (0.02 seconds). This is equivalent to monitor continuously the condition.

It is legal to have a whenever inside another whenever. In this case we should consider the inner whenever as a sub-process of the extern whenever. For example, the following structure:

```
Whenever....
        Whenever....
                Whenever....
                        Whenever....
                        EndWhenever
                EndWhenever
        EndWhenever
        Whenever
        EndWhenever
EndWhenever
```

Can be seen as a tree of processes and sub-processes:



If during the execution of the code of one whenever another one is encountered then the former awakes the latter and gets asleep until the newly activated process decides to stop itself. There is a special clause to stop a sub-process:

Done_whenever(n **);**

Where n indicate the number of levels in the process tree that we want to skip (1 to return the control to the previously active process, 2 to the previous of the previous and so on).

2.7 Instructions

In this section we will describe the instructions that can be normally used in any code part of a BL program (for example in functions or inside whenevers).

Expressions

Any correct expression (see section 2.3) is considered a correct instruction. Obviously if the expression does not include any assignment (or seft increment or decrement) the expression will have no effect.

Conditional instructions

The two conditional constructions permitted in BL are:

- If: The simple conditional typical of all the programming languages:
  If expression **then** Code [**else** Code] **EndIf;**

  Ex:

```
If a>b then
        big=a; small=b
else
        big=b; small=a;
EndIf;
```

- Cond: The multiple conditional:

  Cond cond_expression
  Case expression$_1$ **do** Code$_1$ **EndCase**
  ….
  Case expression$_n$ **do** Code$_n$ **EndCase**
  [**default** Code_default]
  EndCond;

  That searches the first expression in the case list that is equal to the cond_expression and executes the associated code. If such an expression does not exist then the Code_default is executed (if we have defined it).

  The cond_expression must be an integer expression (or a similar one: char,...).

  Ex:

  ```
  Cond state
      Case 0: printf("0"); state=1; EndCase;
      Case 1: printf("1"); state=0; EndCase;
      Default printf("Unknown state");
  EndCond;
  ```

Loop instructions

In our version of BL there are three ways to set up a loop:

- For: With the syntax:

  For index variable **=** initial value [**dec** step] [**inc** step] **to** final value **do**
  Code
  EndFor;

  Index variable should refer to an integer or char variable. The semantic of this construction is to assign the initial value to the index variable and then repeat the sequence:

  - Execute code
  - Increment or decrement the index in step units

  until the final value is reached. It is not possible (nor sensible) to use both clauses 'dec' and 'inc' in the same for. If none of them is used the step is assumed to be "inc 1".

  For example, to initialise an array with n (n>0) elements we can write:

  ```
  For i=0 to n-1 do
      table[i]=0;
  EndFor;
  ```

- Repeat:

  Repeat
  Code
  Until expression**;**

  That repeats the code until the expression evaluates true (different from zero).

  The previous example could be re-written as follows:

  ```
  i=0;
  Repeat
      table[i]=0;
      i=i+1;
  Until i<n;
  ```

- While

While expression **Do**
    Code
EndWhile;

That keeps doing 'Code' while expression is true (evaluation different from zero). To initialise an array using this construction we should do:

```
i=0;
While i<n do
    table[i]=0;
    i=i+1;
EndWhile;
```

2.8 Macros

To facilitate the programming process, our BL compiler includes the definition and use of macros. The syntax to define a macro is:

Macro macro name (parameter$_1$**,**....**,**parameter$_n$**)**
    Any BL Construction using the parameters.
EndMacro

To use it we simply write:

macro name (expression$_1$**,**...**,**expression$_n$**);**

The macros are handled by a pre-compiler that recognises the macro definitions and the use of macros copying the BL construction of the macro definition in the place of the macro use and replacing the parameters of the definition by the corresponding expressions of the use.

One limitation in the use of macros is that the expressions used as macro parameters can not include other macro use.

Using the macro mechanism we can, for example, define the Delay operator (that was a pre-defined action in Brooks's original BL) that stops the current whenever for a specific amount of time:

```
Macro Delay (period)
    Whenever 1 with time period do
        Done_Whenever(1);
    EndWhenever;
EndMacro
```

If we write

```
Delay(0.5);
```

in our program it would be equivalent to have written the following code:

```
Whenever 1 with time 0.5 do
    Done_Whenever(1);
EndWhenever;
```

that sleeps the current process for 0.5 seconds.

2.9 Debugging

Our version of BL compiler includes a utility that allows us to monitorize the use of an input or output port during the program execution. To do this we should use the construction:

Debug port$_1$**,**...**,**port$_n$

anywhere in the root module of our program. Every port has to be specified as follows:

(behavior$_i$,port name$_i$)

When this clause is used then the C program generated by our BL compiler includes code to capture all the operations performed over the listed ports and once the program execution finished a file (with extension DBG) is created that contains all that information. Then we can use the DEBUG program with the syntax:

Debug file_name.DBG

to create an easy to interpret representation of the information contained in the DBG file.

2.10 Comments

Our BL compiler accepts two kinds of comments inside a program:

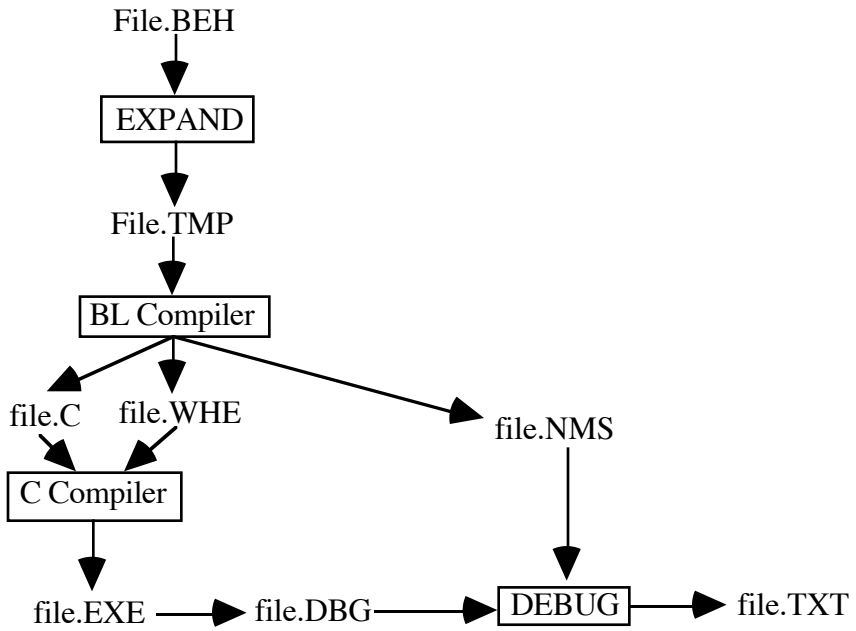
- Line comments: Everything after a '//' and until the next carry return is skiped by the compiler.
- Block comments: Everything inside a /*...*/ is skiped by the compiler.

3 User Interface

Once we know how to produce a Behavior Language program we must explain how to compile that program. Currently there is only one way to invoke the compiler and it is from the command line in the DOS command interpreter (in a near future he hopes we will develop a complete Behavior Language environment). In the next two sections we describe the modules that compose the compiler, the auxiliary files generated and some examples of how to use the compiler.

3.1 The Compiler

Our BL compiler is structured in a pipeline of programs as shown in the next figure:



Next we describe each one of the elements of this pipeline:

- File.BEH: The file containing the BL program.
- EXPAND: A pre-compiler that processes the macros, the includes (see section 2.2.1) the comments and insert information useful for the BL compilation
- File.TMP: The file containing the result of the EXPAND program.
- BL compiler: Our compiler of Behavior Language.
- file.C: A program in C that is semantically equivalent to our BL program.
- file.WHE: A support file for file.C necessary for the C compilation.
- file.NMS: A list with the name of all the ports of the program (including behavior name and port name). This file is useful to generate an easy to understand output when we are debugging a program.
- C Compiler: A standard C compiler. It uses file.C and file.WHE plus other complementary files (for example a C program to control the robot,...) to generate an executable file. We use a Borland Turbo C++ compiler but any C compiler will do.

- file.EXE: The executable file generated by the C compiler. We can execute it simply typing 'file' in the DOS command line. When executed, if we have used the clause DBG in our BL program, it creates file.DBG. This file includes information of all the operations related with the port we are monitorizing that has occurred during the program execution.
- DEBUG: A special program to interpret the contents of the file.DBG file.
- file.TXT: The output of the DEBUG program.

To facilitate the use of all these modules we have implement a small batch program that hides all that process. Its use is:

Compile file.BEH

If there is no error, when Compile finishes the files 'file.EXE' and 'file.NMS' (if debug is turned on) are available. If there are errors (detected either by EXPAND, BL or the C compiler) they are displayed on the screen.

3.2 Examples

Consider that we have a program composed by eight layers (Constants, Hold, Balance, Adaptation, Compliance, Cliff, Walk and Drive)[2], each layer is in a separated file and we want to compile the top layer (Drive). The command that we have to introduce in the DOS command line is:

Compile drive

Inmediately the following messages will appear on the screen:

```
MACRO EXPANSION
BL COMPILATION
STARTING BL COMPILATION .....
>COMPILING: drive.BEH
    >COMPILING: walk.BEH
      >COMPILING: cliff.BEH
        >COMPILING: complian.BEH
          >COMPILING: adaptat.BEH
            >COMPILING: balance.BEH
              >COMPILING: hold.BEH
                >COMPILING: const.BEH
                >const.BEH COMPILED
              >hold.BEH COMPILED
            >balance.BEH COMPILED
          >adaptat.BEH COMPILED
        >complian.BEH COMPILED
      >cliff.BEH COMPILED
    >walk.BEH COMPILED
>drive.BEH COMPILED
BL COMPILATION FINISHED:
          ERRORS: 0
        WARNINGS: 8
STARTING C COMPILATION .....
```

[2]This example is based in a real program for Genghis. For a more detailed description of this program see [3] or [4].

C COMPILATION FINISHED: DRIVE.EXE CREATED

Informing of how the compilation is going on. When the last message appears on the screen the following files are available:

- Drive.EXE: The executable file.
- Drive.NMS: A file with inputs and outputs names.
- Three error/warning files:
  - EE: Contains the expansion error.
  - EBL: With the BL compilation errors and warnings.
  - EC: With the C compilation errors and warnings.

If the program has any error 'Compile' will inform about it and will display the corresponding error file (EE, EBL or EC). In our example no error is detected so EE is empty and EBL and EC contain information about warnings.

The EBL warning messages has the format:

W number**: Warning at line** XX **of file** FF
Problematic line
Warning message

And the error messages are as follows:

E number**: Error at line** XX **of file** FF
Problematic Line
Error message

In our example 8 BL warning messages are generated. One of them is:

W 4: Warning at line 89 of file const.BEH
printf("\n");
Use of an unknown identifier (printf)

Printf is a C function available in BL thanks to the use of the lib clause so BL does not know how to treat it and informs about that problem. If there were no lib clause in the program this warning would have been an error message.

The EC message format depends on the C compiler we use. With the Borland Turbo C++ compiler a typical warning message is:

Warning 5706 drive.c: Unreacheable code in function _M76_primer_pas

That informs of the line and file that produces the warning, the cause of the warning (Unreacheable code in this case) and the name of the function that includes that line. 'primer_pas' is a user defined behavior and '_M76_' is a prefix included by the BL compiler.

Once we succeed in solving all the errors of our program and we are sure that none of the warnings is really dangerous, we can test the program on the robot:

- Make sure that the robot is well conneted either to the current and to the PC.
- Switch the robot on.
- Type

  program name

  in the DOS command line. In our example, we should type:

  drive

4 References

[1] Brooks, R. A. *"The Behavior Language; Users Guide"* Massachussetts Institute of Technology, Artificial intelligence Laboratory, A. I. Memo 1227, 1990

[2] Brooks. R. A. *"A Robust Layered Control System for a Moblie Robot"* IEEE Journal of Robotics and Automation, RA-2, 1986

[3] Celaya, E., Porta J. M. *"Control of a six -legged robot walking on abrupt terrain"* Proceedings of the 1996 IEEE Conference on Robotics and Automation, 1996

[4] Celaya E. Porta J.M., *"Descripció dels comportaments del Genghis"* Technical Report, Institut de Cibernètica, IC-DT-9602, Universitat Politecnica de Catalunya, Spain, 1996 (In Catalan)

Appendix A: The C program

As we have said, our BL compiler generates a C program that once compiled by a C compiler produces the executable file that controls the robot. In this appendix we will briefly describe the components of this C program.

The C program has four parts:

- Initial declarations:
  - Some includes:
    Stdio.h: To use some standard C functions.
    Genghis.C: A program to control the comunication with Genghis through serial line. It also includes some global variable declaration.
    program_name.WHE: A structure that indicates for each whenever when it was last executed and witch sub-whenever is active (see section 2.6 for more details).
  - A temporal buffer declaration: It is used to store temporal data during message passing. It size limits the size of the passed message.
  - STOP_GENGHIS: The global variable declaration.
  - The header of two functions (output and inhibit). They can appear anywhere in the program.
- Program traslation: This part contains the traslation of all the files that conform the user program written in BL. Each construction in BL has an equivalent constructions in C:
  - Lib: Is translated to a C include clause. For example
    lib “conio.h”
    switches to
    #include “conio.h”
  - Constants: Constant declaration are translated using #define in C:
    constant a=4;
    becomes
    #define _M2_a 4
    As you can see, the BL compiler changes the name of the constant adding a prefix. This is because BL programs are organized in independent blocks (see section 2.1) while C programs have just one block. So BL utilizes prefixes in all names (constants, types, variables, modules, behaviors,...) to determine the module (or behavior) that includes that definition. This allows the C compiler to respect the visibility rules of BL. All prefixes have the same form:
    M(module or behavior number)
    The BL compiler is in charge of assigning a different number to each module and behavior (for example the root module has the number 2).
  - Types: BL types becomes a C type using the ‘typedef’ construction:

type uchar: array[5] of unsigned char;

is converted to:

typedef unsigned char _M2_uchar[5];

Or

type complex: record a:double; b: double endrecord;

Becomes

typedef struct { double a; double b;} _M2_complex;

- Variables: BL variables translates into C variables. For example

var a:array [5] of unsigned char={0,0,0,0,0};

In C is:

unsigned char _M2_a[5]={0,0,0,0,0};

- Modules: Each module transforms in a single function that simply calls the functions of the behaviors and sub-modules of that module. In any of these behaviors or sub-modules have an activation rule then the activations condition is tested before calling the corresponding function.
- Behaviors: Each BL behavior correspons to a C function. This function contains the translation of all the reactive rules of that behavior one after another. In this way the parallel execution of the rules becomes a psedo-parallel execution (like it was in the original BL compiler).
- Inputs: The input registers of the behaviors translates into a struct in C. This struct contains information about who, when, with what priority and with what period has changed the register value. It also inform about if the port is inhibited (and if yes, until when) and if the port is additive. Of course, this struct also contains the present register value.
- Whenevers: Each reactive rule translates into a C code that implements the following algorithm:
  - Determine what proces or sub-process is active (see section 2.6)
  - Once determined, see if this process or subprocess activation condition holds. This activation condition contains two parts:
    A) See if it is time to check the Whenever condition again
    B) If yes, check the whenever condition
  - If both parts holds, then the process (or sub-process) code is executed.

  There is an auxiliar data strucuture to determine witch process or sub-process of each whenever is active and when is time to check the whenever condition again. This structure is the contents of the file with extension WHE that the BL compiler cretates.

• Connections: After the program translation there are two structures to control the communications between behaviors. The first of these structures contains informations about each connection (shorted by origin). This information concerts the destination of the connection, the priority and if the user wants to monitorize the use of this connection. The second structure has information of each output. This information includes whether the output is inhibited or not, whether the user

wants to monitorize the output and to witch connections (of the former structure) have this output as an origin.

- Include the output.c file: This file contains the definition of the output and the inhibit functions whose headers appears at the beginning of the program. These functions operates over the connections, outputs and inputs structures defined across the program.
- Main function: The last part of the C program is the main function. In this fucintons the following steps are performed:

  A) Call to all the initializing functions defined by the user in the BL program.

  B) Initialize the serial connections with Genghis.

  C) Main loop: While the robot is not stopped call the modules and behaviors defined in the root module (if its corresponding activation rule holds).

  D) Closes the serial link with Genghis.

  C) Shows some information about the program execution.

  E) If there is some debug information collected during the execution, creates the DBG file. This file can be interpreted with the DEBUG program.

Appendix B: BL Warnings and Errors

BL Warnings

- Use of unknown identifier (X)

  The indentifier X has not been declared in the current program before it has been used. X can be a function defined in a C library (included in the BL program using the lib clause).

- Definition of a never ending whenever

  The user has defined a internal whenever (a whenever defining a sub-process) without a way to exit of this whenever. That is, the whenever does not include any Done_Wnenever.

- Unnecessary specification of PRIORITY in clause DEBUG

  When using a DEBUG clause to state the port we want no monitorize during the program execution we have to specify the origin and destination of the connexion but not the priority.

- Unknown port (B,P) in DEBUG instruction

  The port P of the behavior B does not exist but the user has mentioned it in a DEBUG clause.

BL Errors

INTERNAL ERRORS

- FATAL ERROR: Not enough memory

  BL compiler internal error.

- FATAL ERROR: Symbolic space so small

  BL compiler internal error.

- Connections lost in compilation

  Internal Error in the storage of connections.

PROBLEMS WITH FILES

- File X cannot be opened

  There is a problem when trying to open the file X. It is possible that X does not exist in the current directory (if X is an input file) or that there is not enough space to create a new file (if X is an output file).

PROBLEMS WITH INPUTS AND OUTPUTS

- X is not an output

  X has been mentioned as an output but it does not exist.

- X is not an input

  X has been mentioned as an input but it does not exist.

- X is not a port

  The user is trying to inhibit the (input or output) port X but it has not exist.

- Mismatch priority declaration

  The user has re-declared a connection (for example, using a clause FROM to define a connection previously defined with a clause TO) with a different priority.

- Wrong type in the definition of an additive input

  Additive inputs must be integers, chars or doubles.

PROBLEMS WITH CONNECTIONS

- You cannot connect ports with different type

  The origin and destination of a connection must have the same type.

PROBLEMS WITH IDENTIFIERS

- X is not a type identifier

  X is used as a type identifier but it has not been declared.

- X is not a function identifier

  X is used as a function but it has not been declared.

- X is not a variable

  X is used as a variable but is has not been declared.

- X is an already defined identifier

  The user is trying to use X in a declaration (a constant, a variable, a function,...) but X has already been used previously.

- Undeclared identifier X

  The user has user X as a variable or constant but X is has not been declared.

- Already defined behavior (X)

  X has been used twice as a behavior identifier.

PROBLEMS WITH SPECIAL INSTRUCTIONS

- X is not a constructions allowed in this place

There are some instructions that can not be used everywhere but are reserved for special situations:

- Received: The received must be used inside whenevers. For example they can not be used in functions.
- Whenever: They can be used inside behaviors but not in fuctions.
- Done_Whenever: They must be used in whenevers.
- Output: They can be used in whenevers.
- Inhibit: They can be used in whenevers.
- Return: They can appear just in functions.

- DONE_WHENEVER must jump over one or more WHENEVERS

  The functions DONE_WHENEVER must have a parameter greater than 0.

- Too many activation rules

  Every module or behavior can have one (but not more) activation rule.

PROBLEMS WITH FUNCTIONS

- Wrong number of parametres in the use of function X

  We are using the function X with more or less parameters than the number of parameters of the declaration.

- Wrong return type

  A functions can not return an array.

- The function X does not return anything

  We have declared a function, X, that does not include any return clause but is supossed to return something.

MISCELANIA

- Monostables must be initialized

  When we declare a monostable variable we must indicate its period.

- There is a unbalanced comment

  There is a comment openned but not closed or biceversa.

Appendix C: The BL grammar

In this section we describe formally the grammar of our version of BL in the YACC format (words in uppercase and inside ‘ ‘ are terminal items):

```
program : program_part  program
        |
program_part : LIB constant_string
             | DEBUG port_list
             | INITFUNCTION code ENDINIT
             | CONNECT ‘(‘ identifier ‘,’ identifier ‘)’ TO port_list
             | define_constant
             | define_type
             | define_variable
             | define_function
             | define_module
             | define_behavior

define_constant : CONSTANT identifier ‘=‘ expression ‘;’

define_type : TYPE identifier ‘:’ deft ‘;’
deft : ARRAY ‘[‘ integer ‘]’ OF deft
     | RECORD record_list ENDRECORD
     | ptr_list type_name
record_list : record_element record_list
            |
record_element : identifier ‘:’ deft ‘;’
ptr_list : POINTER TO ptr_list
         |
type_name : | DOUBLE
            | sign INTEGER
            | sign CHAR
            | identifier
sign : UNSIGNED
     | SIGNED
     |

define_variable : VAR identifier extended_type inicialization ‘;’
extended_type : ‘:’ deft
                | MONOSTABLE
                |
inicialization : ‘=‘ value
               | ‘(‘ period ‘)
               |
value : expression
      | ‘{‘ value_list ‘}’
value_list : value ‘,’ value_list
           | value
period : real
       | integer

define_function : FUNCTION identifier ‘(‘ parameter_list ‘)’ return_type
                  local_variables
```

```
                    BEGIN
                    code
                    ENDFUNCTION
parameter_list : parameter parameter_list
               |
parameter : identifier ':' deft
local_variables : define_variable local_variables
                |
return_type : ':' deft
            |

define_module : MODULE identifier
                module_part_list
                ENDMODULE
module_part_list : module_part module_part_list
                 |
module_part : define_constant
            | define_type
            | define_variable
            | define_function
            | define_module
            | define_behavior
            | CONNECT '(' identifier ',' identifier ')' TO port_list
            | ACTIVATION expression ';'

define_behavior : BEHAVIOR identifier
                  inputs
                  outputs
                  behavior_part_list
                  PROCESSES
                  rule_list
                  ENDBEHAVIOR

inputs : INPUTS additive identifier port_type inicialization origin
       |
additive : ADDITIVE
         |
port_type : ':' type_name
          |
origin : FROM port_list
       |
port_list : port port_list
          | port
port : '(' identifier ',' identifier priority ')'
priority : PRIORITY integer
         |

outputs : OUTPUTS identifier port_type destination
destination : TO port_list
            |

behavior_part_list : behavior_part behavior_part_list
                   |
```

```
behavior_part : define_constant
                | define_type
                | define_variable
                | define_function
                | ACTIVATION expression ';'

rule_list : rule rule_list
          |
rule : WHENEVER expression rule_period DO
       code
       ENDWHENEVER ';'
rule_period : WITH TIME expression
            |

code : instruction ';' code
     |

instruction : rule
            | DONE_WHENEVER '(' integer ')
            | OUTPUT '(' identifier ',' expression period ')'
            | INHIBIT '(' identifier ',' identifier period ')'
            | TRIGGER '(' identifier ')
            | RETURN '(' expression ')
            | expression
            | IF expression THEN code else ENDIF
            | COND expression case_list default ENDCOND
            | FOR identifier '=' expression step TO expression DO code ENDFOR
            | REPEAT code UNTIL expression
            | WHILE  expression DO code ENDWHILE
            | identifier '(' expression_list ')'

period : ',' expression
       |

else : ELSE code
     |

case_list : case case_list
          | case
case : CASE expression DO code ENDCASE
default : DEFAULT code
        |

step : INC integer
     | DEC integer
     |

expression : NOT expression
           | '-' expression
           | '+' expression
           | '++' expression
           | '--' expression
           | expression '++'
           | expression '--'
```

```
                | '*' expression
                | '&' expression
                | expression '=' expression
                | expression '*=' expression
                | expression '/=' expression
                | expression '%=' expression
                | expression '+=' expression
                | expression '-=' expression
                | expression '&=' expression
                | expression '^=' expression
                | expression '|=' expression
                | expression '<<=' expression
                | expression '>>=' expression
                | expression '==' expression
                | expression '<>' expression
                | expression '<' expression
                | expression '>' expression
                | expression '<=' expression
                | expression '>=' expression
                | expression AND expression
                | expression OR expression
                | expression '+' expression
                | expression '-' expression
                | expression '/' expression
                | expression '%' expression
                | expression '>>' expression
                | expression '<<' expression
                | expression '&' expression
                | expression '|' expression
                | expression '^' expression
                | '(' expression ')'
                | expression '.' expression
                | expression '[' expression ']'
                | identifier parameters
                | RECEIVED '(' identifier ')'
                | constant
                | CLOCK
                | STOP_GENGHIS

parameters : ('  expression_list ')'
                |

expression_list : no_empty_expression_list
                  |

no_empty_expression_list : expression ',' expression_list
                           | expression

constant : integer
           | '#x' hexa
           | real
           | string
           | char
```

```
identifier : letter dl_list

dl_list : digit
        | letter
        | '_'
        |

letter : 'A'
       | 'B'
       | 'C'
       ……
       | 'Z'
       | 'a'
       ….
       | 'z'

digit : '0'
      | '1'
      ….
      | '9'

integer : digit integer
        | digit

real : integer real_part exponent

real_part : '.' integer
          |

exponent : e s integer
         |

e : 'E'
  | 'e'

s : '+'
  | '-'
  |

hexa : digit_hexa hexa
     | digit_hexa

digit_hexa : digit
           | 'A'
           | 'B'
           | 'C'
           | 'D'
           | 'E'
           | 'F'

string : "" string_character_list ""

string_character_list : string_character string_character_list
                      |
```

and string_character is any available character but “ except if it is preceded by \

char : ‘‘‘ character ‘‘‘

and character is any available character

There are tree especial constructions that can appear anywhere in the program:

- Macro definition

  macro_definition : MACRO identifier ‘(‘ identifier_list ‘)’
  macro_contents
  ENDMACRO

  identifier_list : no_empty_identifier_list
  |

  no_empty_identifier_list : identifier ‘,’ identifier_list
  | identifier

  And the macro contents can be anything using the parameters of the identifier_list

- Macro use

  identifier ‘(‘ expression_list ‘)’ ‘;’

  The expression_list should have as many items as the identifier_list of the macro declaration. During the macro expansion, the macro use is replaced by the macro contents but each expression substitutes the corresponding identifier.

- Coments

In our version of BL there are two kind of comentaries. Line comentaries:

line_comentary : ‘//’ anything

and block comentaries

block_comentary : ‘/*‘ anything ‘*/’

In the block comentary we have to be very carefully with the use of ‘/*‘ and ‘*/’ since if they are not perfectly matched, the can confuse the compiler producing an error.

Appendix D: Reserved words

As in almost all programming languages, BL has some special words that can not be used as an identifiers because the are used in the language constructions. This section is an alphabetical list with this reserved words.

ACTIVATION
ADDITIVE
AND
ARRAY
BEGIN
BEHAVIOR
CASE
CHAR
CLOCK
COND
CONNECT
CONSTANT
DEBUG
DEC
DEFAULT
DO
DOUBLE
ELSE
ENDBEHAVIOR
ENDCASE
ENDCOND
ENDFOR
ENDFUNCTION
ENDIF
ENDINIT
ENDMODULE
ENDRECORD
ENDWHENEVER
ENDWHILE
FOR
FROM
FUNCTION
IF
INC
INHIBIT
INITFUNCTION
INPUTS
INTEGER
LIB
MODULE
MONOSTABLE
NOT
OF
OR
OUTPUT
OUTPUTS

PRIORITY
PROCESSES
RECEIVED
RECORD
REPEAT
RETURN
SIGNED
STOP_GENGHIS
THEN
TO
TRIGGER
TYPE
UNSIGNED
UNTIL
VAR
WHENEVER
WHILE

Appendix E: Genghis Interface

When using the BL compiler with Genghis, there are some functions available to acces sensor data and to command motors. All the parameters and outputs of these functions are "unsigned char" but usually they values do not exceed 127 (so they can be considered "signed char").

- read_force(motor_id)

  Returns the reading of the force sensor of the motor. Due to Genghis internal delays and serial line delays the reaction to each motor movement appears about 0.1 seconds after the movement execution.
- read_irs(irs_id)

  Returns the active infrared sensors reading. The irs_id are:

  0 for the front left
  1 for the front right
  2 for the rear left
  and 3 for the rear right
- read_whiskers(whisker_id)

  Returns the whisker readings with 1 meaning touch detected. Left whisker identifier is 0 for the left one and 1 for the right.
- read_pyro(pyro_id)

  Returns the pyro sensor reading. Pyro_id range from 0 (the left one) to 4 (the right one).
- read_battery_electronics()

  Returns the current electronic battery level.
- read_battery_motors()

  Returns the current motor battery level. The minimum motor battery level is 42. Under that level the robot stops.
- read_mode()

  Returns the mode configuration (four bits).
- read_inclinometer()

  Returns the inclinometer reading. When the robot is horizontal the inclinometer readings ranges around 90.
- read_fsr_force(fsr_id)

  Returns the variable resistance readings. The fsr_id is 0 for the front fsr, 1 for the middle and 2 for the rear one.
- read_fsr_pos(fsr_id)

  Returns the position for the coctact detected by the fsr sensors. The fsr_id are the same as before.
- write_leds(new_leds_value)

  Changes the configuration of leds switched on. The new_leds_value is a 8 bits data where each bit controls a different led.
- read_leds()

  Returns the last current leds configuration.
- write_buzzer(new_sound)

  Changes the sound emited by Genghis. 0 means no sound.
- read_buzzer()

Returns the current Genghis sound.

- write_position(motor_id, new_position)

  Changes the current motor position for a new_position. The parameter motor_id allows to choose the motor to move.

- read_pos(motor_id)

  Returns the current position (that is the last command) of the selected motor.

Data from motor force sensors and whiskers and commands to motors, leds and buzzer can change every 0.02 seconds and the rest of data changes every 0.04 seconds.

There are some pre-defined constants to identify the motors:

alfa0, alfa1, alfa2, alfa3, alfa4, alfa5
beta0, beta1, beta2, beta3, beta4, beta5