LEGO P-brick
Script Code Language

Table of contents

1 LEGO P-brick Script code specification

LEGO P-brick Script will be used as the implementation language for the graphical programming environment to be provided as part of the RIS2.0 Set, to implement other graphical environments and as a starting point for future scripting languages. It will be based upon the graphical programming paradigm that has evolved from RIS and stand-alone Scout design: Sequential execution of a main program stack, together with watchers that continuously monitor events and react to these events by executing additional stacks in parallel with the main stack. Conflicts on access to shared resources can be resolved using the priority based access control system.

1.1 Program structures

A LEGO P-brick Script program consists of an optional main stack, named event watchers and/or named or numbered tasks, together with macro definitions. Watchers monitor one or more system or user defined events, and execute a stack when an event is received. Tasks contain a stack and can be started and stopped from other stacks.

Watchers can be declared so that, once stack execution has been triggered, the stack runs to completion (the default) even if other events monitored within the same watcher are detected. Or stack execution can be interrupted (and restarted) when an event in the watcher eventlist occurs. If a triggered watcher stack runs to completion, events may be missed.

Watchers and tasks are named (or numbered) so they can be started and stopped. Initially all watchers and tasks are stopped. Once started, a watcher can be level triggered using the trigger command. The trigger command fires selected events if the immediate value of the event source satisfies the condition of the event. The fire command can be used to generate an event (user or system) or list of events immediately.

Each event watcher will be implemented as a task. If there are any user defined events, a dispatcher task will be allocated to continuously monitor event source values. The dispatcher task will fire these as system events, so the watcher task code will be identical for user and system events. Program example:

```
program MyProgram {

  #include <MyMacros.h>                    // contains macro definitions

  sensor leftTouch on 1
  sensor Opto on 2
  sensor rightTouch on 3

  leftTouch is switch as boolean
  Opto is light as percent
  rightTouch is switch as boolean

  event gloomy when Opto is 0..20          // dispatch loop events
  event dull when Opto is 30..40
  event intense when Opto is 80..100
  event leftPressed when leftTouch.pressed      // system events
  event rightPressed when rightTouch.pressed

  main {
    start feelerWatcher
    start lightWatcher
    start MyTask
    trigger gloomy, dull, intense          // initial level triggering
  }

  watcher feelerWatcher monitor leftPressed, rightPressed {
    if leftPressed {avoidRight}            // call to macro
    if rightPressed {avoidLeft}
  }
```

```
   watcher lightWatcher monitor gloomy, dull, intense {
     if gloomy {stack}
     if dull {stack}
     if intense {stack}
   } restart on event                  // sensor events will re-trigger the
                                       // watcher during execution

   task MyTask {
     stack
   }
}
```

Global & local declarations, and scope

Global variables, timers and counters must be declared before the main block. Any initialization of global variables will be placed at the start of the main task. Local variables can only be declared within the main block, watchers, tasks and macros. Macro formal parameter names are treated as local variable declarations. Constants (including sensor and output port assignments) can be placed before the main block, or at the beginning of the main, watcher, task or macro blocks. Usually these declarations will be included as headers at the start of the program.

Global variables are allocated from P-brick global variables. Local variables declared within main, watchers, tasks, macros, or as macro formal parameters, are allocated from task specific local variables.

Local declarations of variables (in main, watchers, tasks or macros) hide globals with the same name:

```
program testVars {
  sensor left on 1
  sensor right on 2
  event leftPressed when left.pressed
  event rightPressed when right.pressed
  var y = 100
  macro ping(y) {
    tone y for 10
  }
  macro pong {
    local y = 110
    tone y for 10
  }
  main {
    local y = 220
    tone y for 100 wait 100
    ping(660) wait 50 pong
    start beep start bop
  }
  watcher beep monitor leftPressed {
    local y = 440
    tone y for 50
  }
  watcher bop monitor rightPressed {
    local y = 880
    tone y for 50
  }
}
```

Local variables cannot be declared or used in immediate commands, so expressions requiring temporary variables cannot be evaluated. For example:

```
sensor opto on 3
var v = (10 * (1020 - opto.raw)) / 102
get v  //illegal
```

But you can hand-code the same expression using globals:

```
sensor opto on 3
var v = opto.raw
v *= -1
v += 1020
v *= 10
v /= 102
get v  //get light %
```

Program structures are:

```
program programname { program body }
```

Inside the program body we have compiler directives and declarations followed by:

```
macro macroname (parameterlist) { command stack }

main { command stack }

watcher watchername monitor eventlist { command stack } [restart on event]

task taskname { command stack }

fragment (x, y) { command stack }

comment (x, y) "Any literal string all on one line"
```

fragment is used to frame a floating group of blocks in a graphical programming environment. (x, y) gives the position of the floating stack.

comment is used to insert floating comment posters in a graphical programming environment. (x, y) gives the upper left corner of the poster square. Inside the double quotes any string will go.

/r	Carriage return
/n	New line
/t	Tab
/"	Double quotes

Watchers, tasks and main are started and stopped (main is also started and stopped when you press the RUN button):

```
start watchername
stop watchername

start taskname
stop taskname

start main
stop main

eventlist        Comma separated list of named events
parameterlist    Comma separated list of the parameters of the macro
```

To stop all tasks from running use:

```
stop tasks
```

1.2 Macros

```
macro macroname (parameterlist) { commands }

parameterlist    Comma separated list of the parameters of the macro
```

Macros are called by name.

Macros can be downloaded to RAM subroutines or the macro code can be expanded inline. The compiler will optimize to reduce code size. Example:

```
program flashBeep {
  var note = 50
  macro flash(times, freq) {
    repeat times {tone freq for 10}
  }
  macro beeps {sound 3 sound 5 sound 1}
  main {
    forever {flash(2, note) beeps note += 5}
  }
}
```

The compiler could decide to implement the 'flash' macro as a subroutine if it is used in several places. The 'beeps' macro might be better inline.

There is a trade-off between the need to initialize local variables before calling subroutines, and the inline code size.

The compiler will need to take care with macro parameters that are, for example, sensor values, when implementing macros inline.

You can call another macro from within a macro. A macro cannot call itself.

If you want to use port assigned resources inside a macro, these should be passed in the parameter list:

```
program GoForLight {

   output A on 1
   output B on 3
   sensor Opto on 3

   macro GoLight(nLeft, nRight, nLight) {
      if nLight.raw < 700
           {fd [nLeft nRight]}
      else {bk [nLeft nRight]}
      on [nLeft nRight] for 100
   }

   main {
      forever {
      GoLight (A, B, Opto)
      wait 100
      }
   }
}
```

1.3 Control structures

```
if value relop value { commands } [ else { commands } ]
if value is [not] range { commands } [ else { commands } ]
select value {
  when val1 { Commands }
  when val2 { Commands }
  ...
  when valN { Commands }
  [ else { Commands } ] }
while value relop value { commands }
while value is [not] range { commands }
repeat repeatnumber|random a [to b] { commands }
repeat { commands } until eventlist
forever { commands }
wait time|random time1 [to time2]
wait until eventlist
```

value	Number, constant, variable, sensor, timer, counter, message, property
val1-valN	Number, constant or range
relop	= or < or > or <>
repeatnumber	Number or constant
range	v1..v2 vN is number, constant or variable
time	Number, constant, variable
eventlist	Comma separated list of named events

Random a returns a random number from 0 to a both inclusive.

Random a to b returns a random number from a to b both inclusive.

The **If** structure evaluates a condition once. If the result is true, the commands in the first list are executed. If the result is false and there is an **else** list of commands, they are executed.

The **select when** structure compares the specified value with a series of constant numbers or ranges. The first condition to mach is executed. If none of them match an optional **else** part can be executed.

The **while** structure repeatedly evaluates a condition until it is false. While the result is true, the commands in the list are executed.

The **repeat** structure repeats commands for the specified number of times.

The **forever** structure repeats the command block forever.

For explanation of control structures using events se Section 1.4.

1.4 Events and event settings

```
fire eventlist
trigger eventlist
monitor eventlist { commands } [retry, abort, restart, stop] on event
repeat { commands } until eventlist
wait until eventlist
if eventlist { commands } [ else { commands } ] // Only inside watcher stack

calibrate (eventname)
```

```
eventname       Literal alphanumeric name for an event
eventlist       Comma separated list of named events
```

Events are declared with a name and a condition or a link to a system event before use:

```
event leftPressed when leftTouch.pressed
event bright when opto is 80..100
```

System events are properties of declared event sources:

```
var Score = 0
event EndGame when Score.high
watcher Winner monitor EndGame {sound 1 Score = 0}
```

In the table bellow a list is given of the system event sources with their events and values.

Declaration	Events	Values (read only)
sensor <name> on <1 | 2 | 3> <name> is switch as raw, boolean, transition, periodic	pressed, released low, normal, high click, doubleclick	type (1, 5, 6 or 7) raw (0-1023) value (mode dep.)
sensor <name> on <1 | 2 | 3> <name> is temperature as raw, celsius, fahrenheit	pressed, released low, normal, high click, doubleclick	type (2) raw (0-1023) value (mode dep.)
sensor <name> on <1 | 2 | 3> <name> is light as raw, percent	pressed, released low, normal, high click, doubleclick	type (3) raw (0-1023) value (mode dep.)
sensor <name> on <1 | 2 | 3> <name> is rotation as angle	pressed, released low, normal, high click, doubleclick	type (4) raw (0-1023) value (-32768 to 32767)
timer <name>	pressed, released low, normal, high	value (0-32767)
counter <name>	pressed, released low, normal, high click, doubleclick	value (-32768 to 32767)
var <name>	pressed, released low, normal, high click, doubleclick	value (-32768 to 32767)
PB Message (no declaration required)	message pressed, released low, normal, high click, doubleclick	value (0-255)

Source/Event combinations are addressed by a dot notation: <SourceName>.<EventType>.

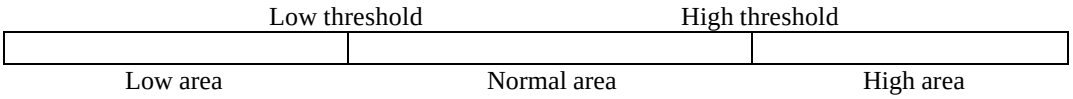
For PB Messages the predeclared message event is thrown every time a message is received. The other event types are addressed by message.<EventType>.

Not all combinations of event sources and events might make perfect sense, but they are available.

A named event has a set of properties, addressed by <EventName>.<EventProperty>

Properties are: low, high, hysteresis
time
state

In general each event has three areas: Low, Normal and High . These three areas are defined by the low and high threshold properties:



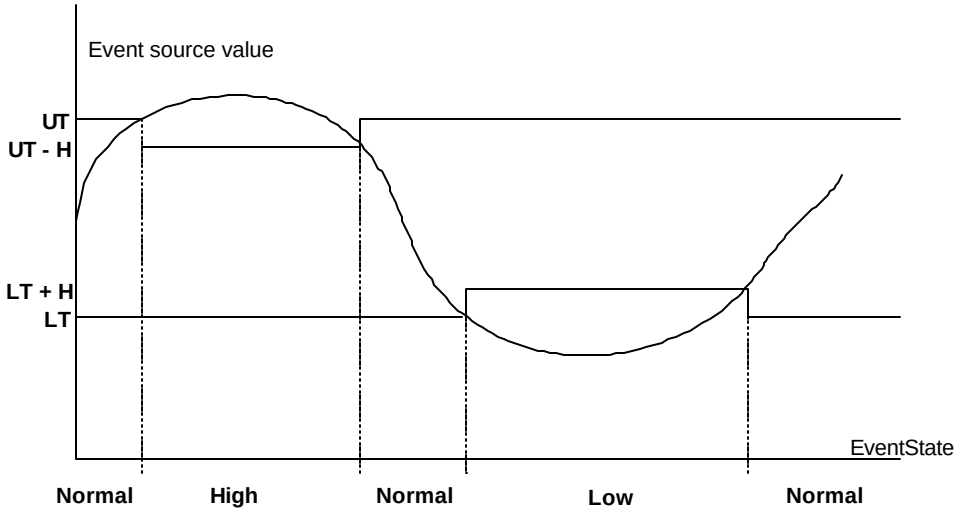
An undefined area, is active at startup.

Depending on the area where the value of an event source is located, the state property of the event will be either low, normal, high or undefined (0, 1, 2, 3 respectively).

State transitions are generated using the low- and high thresholds together with the hysteresis. The conditions for state transitions are as shown bellow.
If (UT < LT + H) the event state will be undefined. If later (UT >= LT + H) state transitions from undefined can take place.

Current State	Condition	New State
High	Value < LT	Low
	Value <= UT – H	Normal
Normal	Value > UT	High
	Value < LT	Low
Low	Value > UT	High
	Value >= LT + H	Normal
Undefined	Value > UT	High
	Value < LT	Low
	Else	Normal

For the message event the state will be undefined whenever the mailbox is empty (message = 0).



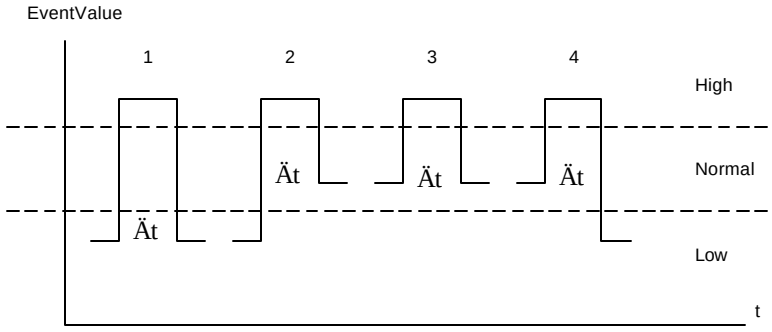
Events are generated on state transitions:

Events	State transitions
Low	High, Normal or Undefined to Low
Normal	High, Low or Undefined to Normal
High	Low, Normal or Undefined to High

Click and double click events are defined from a sequence of certain state changes:

Events	State transitions
Click	A transition into High area and back again
Doubleclick	Two consecutive Clicks

The figure below shows the different ways of generating a Click event.



For the click to be accepted the value of Ät has to be between 50ms and 50ms + ClickTime, where ClickTime is set through the time property of the named event (for light blinks a reasonable value for ClickTime is 150 ms).

A doubleclick is two consecutive clicks with Ät in between.

Pressed and released events are generated using fixed 45%/55% limits.

An eventlist is a comma separated list of named events, used within a watcher, monitor, sound feedback, fire or trigger commands.

Examples of the use of eventlists:

```
sound when leftPressed, bright          //enable sound feedback on listed events
fire leftPressed, bright                //throws listed events
trigger leftPressed, bright             //throws events if event source value
                                        //satisfy event condition
```

The **fire** eventlist command will immediately throw the events given in the eventlist.

The **trigger** eventlist command will test the condition for each event in the eventlist and throw events if the condition is true.

The **monitor** eventlist structure is used to define a code section inside which the operating system of the p-brick will monitor the given list of events. If one of the events in the list happens while executing inside this section the section will be exited and execution will continue in one of 4 different places:

retry on event	Resume at beginning of monitor section
abort on event	Resume at end of monitor section
restart on event	Go to beginning of stack
stop on event	Go to end of stack

Examples:

```
// beep until light goes low
monitor lightLow {forever {sound 1 wait 10}} abort on event

// retry block if switch pressed
monitor leftPressed {on C wait 100 off C} retry on event
```

The **wait until** eventlist structure is an empty monitor that aborts on event.

The **repeat until** eventlist structure is a repeat forever loop inside a monitor that aborts on event.

You cannot nest monitors (remember that wait- and repeat until are also monitors). If you place a try block within a monitor, the only fail options allowed are retry and abort (you cannot use restart or stop):

```
monitor leftPressed {
  forever {try {on A wait 10 off A wait 10} retry on fail}
  } abort on event

repeat {
  try {on B wait 10 off B wait 10} retry on fail
  } until bright
```

Inside a watcher you can test on a subset of the watcher events:

```
watcher MyWatcher monitor leftPressed, rightPressed {
  if leftPressed  {tone 1000 for 100}
  if rightPressed {tone 2000 for 100}
  }
```

1.5 Access control

```
priority prt
try { commands } [retry, abort, restart, stop] on fail


prt             Number or constant (1-8, 1: High priority)
```

The **try** structure is used to define a code section inside which the operating system of the p-brick will perform priority based access control on some of the critical resources used inside the section. These resources are: Motors, sound and VLL.

The compiler will automatically scan each try-block and generate a list of used output resources. If the block does not use any resources, no access control code is going to be generated. The priority can be set at any time.

If a critical section fails to acquire requested resources, or loses resources before completion, the section fails and aborts. Execution continues in one of 4 places:

retry on fail	Resume at beginning of try section (thus try to get access to resources again and redo section)
abort on fail	Resume at end of try section (don't bother to finish the section, just abort and go on)
restart on fail	Go to beginning of stack (restart the entire stack)
stop on fail	Go to end of stack (abort the entire stack (in a Watcher you will return to monitoring for events))

Examples:

```
main {
  priority 6
  sound 1
  try {on C wait 10 off C} retry on fail  // retries block until it completes
  sound 2
}

main {
```

```
  priority 6
  sound 1
  try {on C wait 10 off C} abort on fail  // skip block if try block fails
  sound 2
}

main {
  priority 6
  sound 1
  try {on C wait 10 off C} stop on fail   // stop executing main if try block
                                          // fails
  sound 2
}

main {
  priority 6
  sound 1
  try {on C wait 10 off C} restart on fail      // start main again if try block
                                                // fails
  sound 2
}
```

Try blocks can be used anywhere a command can be used within a stack. Try blocks can therefore be used within other control structures. Using a try block within a watcher stack allows the watcher to immediately continue monitoring events if the try block exe cution fails with stop (or restart). This example gives up (using stop) if it cannot turn output 1 on, and starts monitoring events again immediately:

```
watcher buttonWatcher monitor leftPressed {
  try {on A wait 10} stop on fail
  try {off A} abort on fail
  sound 2
}
```

You cannot nest try blocks. If you place a monitor block within a try block, the only event options allowed are retry and abort (you cannot use restart or stop).

1.6 Sensors

Sensors must be declared by name (thus associated with a specific input port) and assigned a type and mode before the value of a sensor can be accessed (se Section 1.12). Once declared, sensors are always referred to by name. They can be cleared, calibrated and the sensor values can be read:

```
sensor sensorname on port

sensorname      literal alphanumeric name for a sensor
port            number (1, 2 or 3 for RCX input ports)

sensorname is type
sensorname as mode
sensorname is type as mode

type            Unknown, switch, temperature, light, rotation
mode            raw, boolean, transition, periodic, percent
                celsius, fahrenheit, angle
```

Sensor commands are:

```
clear sensorname          // Clear the sensor value (used in transition and
                          // period counter mode) and for rotation sensor
calibrate (sensorname)    // Auto set the high, low and hysteresis pro-
                          // perties of the events that uses this sensor as
                          // source
```

Once a sensor is declared by name and assigned to an input port you access the values of the sensor as properties:

```
Var = Opto                // Loads processed value of Opto into Var
Var = Opto.raw            // Loads raw value of Opto into Var
Var = leftTouch.type      // Loads the type of leftTouch into Var
                          // For a touch sensor this could be the ID
```

The processed value can be of various sorts depending on sensor type and mode.

The raw property gives the AD converted value of the sensor.

The type property holds the type of the sensor. For a switch type this would be the ID of the switch.

There are four types of sensors:

- Switch A passive resistance sensor
- Temperature A passive resistance sensor
- Light An active current sensor
- Rotation An active gray-code sensor

For each sensor type there are several modes:

- Raw The AD-converter value (0-1023)
- Boolean 0 or 1 based on fixed 45%/55% thresholds
- Transition Counts boolean transitions (0 to 1 or 1 to 0)
- Periodic Counts boolean periods (0-1-0 or 1-0-1)
- Percent Converts the raw value to a 0-100% representation
- Celsius Converts a raw value into degrees celsius
- Fahrenheit Converts a raw value into degrees fahrenheit
- Angle Converts the gray-code scale of the rotation sensor into increments or decrements (depending on direction) ±16 counts per revolution

Depending on the sensor mode a sensor value is calculated.

E.g.

SensorType : Switch
SensorMode : Boolean
SensorValue: If switch is pressed:1, if switch is released: 0.

SensorType : Light
SensorMode : Percent
SensorValue: 0-100 giving light value in percent

For a description on the use of sensors as event sources: Se Section 1.4.

1.7 Motor control

Motors must be named before they can be controlled, because they must be associated with a specific output port (se Section 1.12). Once declared, motors are always referred to by name. They can be controlled individually or in a list, in a

"task local" manner or by using the global motor control settings. For each motor you can read the status or the power setting.

```
output name on port

name            literal alphanumeric name
port            number (1, 2 or 3 for RCX output ports A, B and C)
```

Example:

```
output motorA on 1
```

Motor control commands are:

```
on ports
on ports for time|random t1 [to t2]
off ports
float ports
forward ports
fd ports                      // Short form of forward
backward ports
bk ports                      // Short form of backward
direction fdports, bkports    // Combined forward and backward command
dir fdports, bkports          // Short form of direction
reverse ports
power ports pwr|random pwr1 [to pwr2]
```

In addition to commands which directly control motors (on, off, fd, bk, float, reverse, power), it is possible to control global motor settings via the following commands:

```
global forward ports
global backward ports
global direction fdports, bkports
global reverse ports
global on ports
global off ports
global float ports
global power ports pwr|random pwr1 [to pwr2]

ports             Named output e.g. motorA
                  List of ports e.g. [motorA motorB]
time              Constant giving on time in 0.01 seconds
pwr               Constant or variable giving power level (1-8)
```

The "on Ports for Value" command is exactly the same as "on Ports wait Value off Ports". Forward and backward set the output polarity; reverse toggles it. Power, forward, backward and reverse do not change the state (i.e. on, off or float) of the outputs.

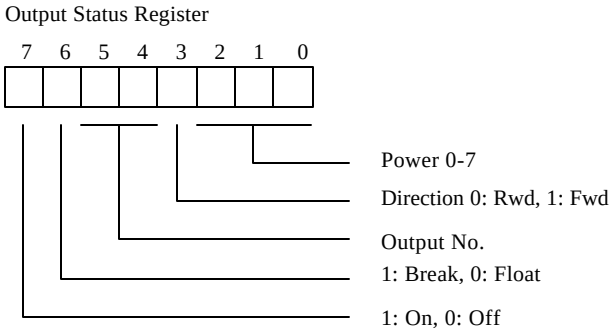
Examples:

```
power [motorA motorC] 8
on [motorA motorC] for random 100 to 200
global off [motorA motorC]
```

To get the current status of an output port use the status property of the port:

```
MyVar = MotorA.status
```

The format of the Output status register is given below:



The direction bit is only used when the output is On.
The break/float bit is only used when the output is Off.

To get the power setting explicitly use:

```
MyVar = MotorA.power
```

1.8 Sound control

```
sound on
sound off
sound soundnumber
tone freq for time
clear sound             // flushes the sound buffer

eventlist       Comma separated list of named events
soundnumber     Constant giving System sound number
freq            Constant or variable giving tone frequency (1-20000)
time            Literal expression giving tone duration in 0.01 seconds up to
                2.55 sec.
```

1.9 Soft resource control

Variables

Variables are declared by name as global or local and can optionally be initialized with a value:

```
var name {= value}            // Allocates a global variable
local name {= value}          // Allocates a local variable

name            literal alphanumeric name
value           constant or number
```

Set variables

You can load a variable with the value of any other operand type or a random number.

```
clear Variable1                     // Set Variable1 to zero
Variable1 = Opto.raw
Variable2 = Variable11
Variable3 = random 10 to 16
```

Math on variables

You can use the usual range of arithmetic (+ - * /) and logic (& |) operators in an expression. Round brackets should be used to make the order of evaluation clear. The compiler will allocate anonymous variables on a per-task basis for use within each expression; once allocated these are reused within each task.

Assignment operators can also be used: +=, -=, *= , /=, &= and |=. These are especially efficient, as no temporary variables are required.

```
Variable1 = (Variable2 + 45)*5
Variable1 = Variable2 & 0x7F
Variable1 += 1

Var = abs ( expression )      //Returns absolute value of expression

Var = sgn ( expression )      //Returns -1 if expression is negative, 1 if
                              //positive
```

Devision by 0 leaves the operand unchanged and abs (-32768) returns 32767.

Timers

```
clear Timer1
```

Counters

```
clear Counter1
```

You can do math on counters just like on variables.

Mailbox control

```
send pbmessage       // from PBrick or from a tower
clear message

pbmessage       Constant or variable (1-255)
```

Datalog control

```
clear data logsize          // allocates and clears a memory block for datalog
log name                    // stores a snapshot of named value in the datalog
get data startadr, count    // returns logged data (direct command only)

logsize         number of bytes in datalog memory area
name            literal alphanumeric name
startadr        start address of block to upload
count           number of bytes to upload
```

1.10 Direct only commands (not downloadable)

Getting information:

```
brick alive?                // returns 1 if brick is alive, otherwise 0
brick version?              // returns string containing ROM and Firmware
```

```
                              // version
brick battery?                // returns battery voltage in milliVolts
get map                       // returns the memory map of the brick
get startadr, count           // returns a block of values from brick memory
get data startadr, count      // returns logged data (direct command only)

startadr          start address of block to upload
count             number of bytes to upload
```

Emulating the remote-control:

```
remote remotecode             // send out a remote control code from tower

remotecode        Remote code bit-mask (se relevant documentation)
```

Deleting Tasks and Subroutines:

```
clear tasks         // delete all tasks
clear task n        // delete a task
clear subs          // delete all subroutines
clear sub n         // delete a subroutine

n                 Task or Sub number
```

1.11 Configuration commands

```
slot slotnumber          // select program slot n (RCX: 1..5)
boot rom                 // put RCX in boot mode ready to receive firmware
boot firmware            // unlocks the firmware
sleep                     // turn off PBrick now
clear sleep              // resets the PBricks sleep timer
sleep after timeout      // turn off PBrick after t minutes
randomize                // re-seeds the random number generator
display name[:decpoint] // selects a value to continuously monitor on the LCD
watch hh:mm              // set internal PBrick watch

brick tx power pwrlevel // set PBrick transmitter power (RCX: 0 or 1)

slotnumber        program slot 1 to 5
timeout           0 means never, 1-255 are minutes
name              literal alphanumeric name for a value
decpoint          selection of decimal point 0: none to 3: 3 decimals
hh:mm             hours and minutes 00:00 to 23:59
pwrlevel          IR transmitter level 0: low or 1: high
```

1.12 Declarations

```
const name = value
var name [= value]             // Allocates a global variable
local name [= value]           // Allocates a local variable
output name on port

name              literal alphanumeric name
port              number (1, 2 or 3 for RCX input- or output ports)
value             constant or number
```

```
sensor sensorname on port
sensorname is type
sensorname as mode
sensorname is type as mode

sensorname      literal alphanumeric name for a sensor
type            Unknown, switch, temperature, light, rotation
mode            raw, boolean, transition, periodic, percent
                celsius, fahrenheit, angle


timer name
counter name [= value]
```

Declaration of events:

```
event name when eventsource[.eventname]
event name when value relop value
event name when value is [not] range

eventsource     Any source of a system event
eventname       A valid eventname for the given eventsource
value           Constant, named variable, sensor, timer, counter, number,
                message
relop           = or < or > or <>
range           v1..v2 vN is number, constant or variable
```

Examples:

```
event Timeout when timer1.high
event Detection when varProximity > 20
event Bright when Opto is 80..100
```

All declarations (except **local** of cause) are global. Reserved words cannot be used for names. Constant declarations can be used to make programs more readable (by providing meaningful names), and also to make it easier to alter program behavior (if their values are used in different places in the program).

Once resources are declared they are used by name reference. Program example:

```
program MyProgram {
// Declarations
const st_Light = 3
const sm_Percent = 4
const LightThreshold = 55
const MyLimit = 50
const MyDelay = 10
output A on 1
sensor Opto on 3
Opto is st_Light as sm_Percent
timer MyTimer

main {
while Opto < LightThreshold { }
   clear MyTimer
   while MyTimer < MyLimit { on A wait MyDelay off A wait MyDelay }
   }
}
```

1.13 Layout and comments

White space, comments & case are ignored.
Multi-line comments (or comments within code) start with /* and end with */.
Single line comments start with // - everything up to the end of line is ignored.

Example:

```
// This is a single line comment
main {
  repeat /* this is a short comment */ 2 {  // this is ignored
    sound 3 wait 100
/* This comment extends
   over two lines */
  }
}
```

The **comment** structure is used to insert floating comment posters in a graphical programming environment. (x, y) gives the upper left corner of the poster square. Inside the double quotes any string will go.

```
comment (x, y) "Any literal string all on one line"
```

/r	Carriage return
/n	New line
/t	Tab
/"	Double quotes

1.14 Compiler directives

Directives for the compiler are (Description will be given):

```
#include <filename>
```

1.15 Sample programs

This program is to be used with the RCX 2.0:

```
program Avoider {
  #include <RCX2.h>
  #include <RCX2MLT.h>

  const BACKTIME   = 100 // 1 second
  const TURNTIME   = 50  // 0.5 second
  const DANCETIME  = 10  // 0.1 second
  const AVOIDLIMIT = 4   // 5 strikes and you're out
  const TICKTIME   = 10  // 1 second

  sensor  LeftTouch  on 1
  sensor  RightTouch on 3
  timer   Timer1
  counter AvoidCount

  event LeftPressed  when LeftTouch.pressed
  event RightPressed when RightTouch.pressed
  event DontBugMe    when AvoidCount > AVOIDLIMIT
  event Tick         when Timer1 = TICKTIME
```

```
  main {
    bbs_GlobalReset([A B C])
    priority 8
    clear timer1
    start Heartbeat
    start AvoidTouch
    start TooMuch
    display AvoidCount
    try {
    forever { bb_Forward(A,C,1000) }
    } retry on fail
  }

  watcher AvoidTouch monitor leftPressed, rightPressed {
    priority 3
    try {
      AvoidCount += 1
      sound 3
      if leftPressed {
        bb_Backward (A, C, BACKTIME)
        bb_SpinLeft (A, C, TURNTIME)
      } else {
        bb_Backward (A, C, BACKTIME)
        bb_SpinRight(A, C, TURNTIME)
      }
    } stop on fail
  } restart on event

  watcher TooMuch monitor DontBugMe {
    priority 2
    try {
      sound 6
      sound 6
      bb_Dance(A, C, 1, DANCETIME)
      clear AvoidCount
      } stop on fail
  } restart on event

  watcher Heartbeat monitor tick {
    priority 7
    clear Timer1
    try {
      tone 36 for 5 wait 10 tone 36 for 5
    } abort on fail
  } restart on event
}
```