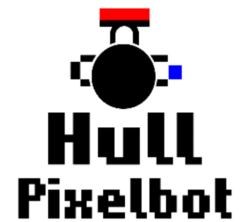



# HullOS Python-ish Specification

Version R1.3

Rob Miles

HullOS is an operating system designed for use in small robots and the Hull Pixelbot. It provides an extensible scripting language that can be used to create programs that are stored inside an Arduino based device. The programs can be downloaded into the device using the serial port and are stored inside the Arduino using its internal EEPROM. When the Arduino is powered on the HullOS program stored in it is automatically executed.

HullOS programs can be written at two levels. At the high level you can use HullOS scripting commands in a language we called “Python-ish”. At the low level you can use HullOS "star" commands. Each star command equates to a single low-level command that is interpreted when the device runs the program. A scripting command may be comprised of one or more "star" commands. A star (*) command provides access to low level programming features. Star commands are pre-ceded by the * character. They can be used at any point in a program. HullOS programs support the following programming constructions

- named variables and simple expressions
- conditional execution using if – else
- nested looping constructions
- direct access to hardware resources
- control over the robot stepper motors
- print messages to the serial port

The size of a program that can be stored in the smaller Arduino devices is limited to a few hundred statements.

## HullOS Direct commands

The robot will accept individual commands at any time, even when a program is running. Some commands can be used to control program execution. These direct commands can also be used in scripts.

### Set the colour of the pixel

You can set the colour of the pixel to one of a number of colours:

```
red
```

The above statement will turn the robot pixel red. The colours that can be used are:

```
red green blue cyan magenta yellow white black
```

A greater range of colours can be selected using the star commands. Take a look at the *PC command if you want more colours.

### Make the robot angry or happy

The rate at which the pixels flash on the robot can be controlled by to program commands:

```
angry
```

This tells the robot to go into "angry" mode. The pixel will flash in an angry way.

```
happy
```

This tells the robot to go into "happy" mode. The pixel flashing will slow down as the robot becomes more relaxed. If you want to create even more subtle robot moods, take a look at the *PF command.

## Move the robot

The `move` command can be used to start the robot moving, and to set a distance for the move. Move commands can also be configured to move in a particular time.

### Starting moving

```
move
```

The above statement will start the robot moving forwards. Note that the robot will continue moving until you give it a different movement command.

### Moving a distance

You make the robot move a particular distance by giving that distance after the move command:

```
move 100
```

The above statement will cause the robot to move 100 mm and then stop.

**Note that the movement may not be very accurate, you can improve accuracy by using the *MV command to tell HullOS the size of your robot wheels and their spacing.**

The move statement can be used to make the robot move backwards by using a negative value:

```
move -200
```

The above statement would move the robot backwards 200 mm.

#### *Moving a distance in a given time*

You can make the robot move a particular distance in a given time by adding an `intime` value:

```
move 100 intime 100
```

The value following the intime keyword is a number of **tenths of a second** that the move should take to complete. The above statement should cause the robot to take ten seconds to move 100 mm.

**Note that if you give a silly time, for example move 1000 millimetres in a one tenth of a second, the robot will not move.**

## Turn the robot

The turn command can be used to make the robot turn. It turns by moving one-wheel forwards and the other backwards, so that it rotates on the spot. If you want the robot to follow a curved trajectory, take a look at the arc command, which is next.

### Starting turning

```
turn
```

This command would cause the robot to start turning clockwise. Note that the robot will continue turning until you give it a different movement command.

### Turning an angle

The turn command can be followed by the value of an angle which the robot is to turn.

```
turn 90
```

This command would cause the robot to turn 90 degrees (a right angle) clockwise. You can turn anti-clockwise by giving a negative value to turn.

### *Turn a distance in a given time*

You can make the robot turn a particular angle in a given time by adding an `intime` value:

```
turn 360 intime 600
```

The value following the intime keyword is a number of **tenths of a second** that the turn should take to complete. The above statement should cause the robot to take 60 seconds to perform a complete rotation.

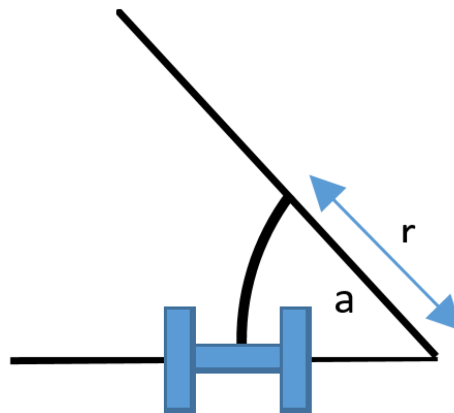
**Note that if you give a silly time, for example turn 1000 degrees in a one tenth of a second, the robot will not turn at all.**

## Move over an arc

This statement allows the robot to move in an arc. This can be a bit hard to understand. Experiment with arcs to understand how they work.

```
arc 0 angle 90
```

The first parameter is the `radius` of the arc, the second is the `angle` giving the angular distance around the arc.




The figure above shows how this works. If the radius is positive the curve will be about a point which is to the right of the robot, and the robot will turn clockwise as it moves. If the radius is negative the curve will be about a point that is to the left of the robot, and the robot will turn counterclockwise. The centre point of the arc is on a line drawn between the two wheels. If the angle is positive the robot will turn clockwise.

In the statement above the radius is 0, i.e. the centre of the arc is the point midway between the wheels. This means that the robot would rotate about its axis.

### *Move over an arc in a given time*

The arc command can be followed with an `intime` value to give the number of **tenths of a second** it should take for the robot to complete the arc.

```
arc 1000 angle 360 intime 600
```

This statement would cause the robot to describe a complete circle of 1000mm radius, and take a minute to do this.

## Make a sound

The robot is fitted with a small speaker that can produce quite annoying sounds. You can use the `sound` command to make these:

```
sound 1000
```

The value following the sound command is the frequency of the sound to be made, in Hz. It is best not to make sounds lower than 100 Hz or higher than about 5000 Hz, although you are free to experiment. The sound quality is not great, but I quite like it. Sounds play for half a second.

### *Play a note*

If you want to play a note in a tune you can follow the sound frequency with a duration. The duration is given in thousandths of a second:

```
sound 2000 duration 1000
```

This statement would play a note of 2000 Hz for a second.

### *Play a note and wait*

The sound command will start a note playing. It will not wait for the note to finish. This can make it hard to play tunes, because you want your program to wait for a note to finish before playing the next one. You can add a wait command to the end of a sound command to tell the program to wait for this note to finish playing before the next statement in a program is performed.

```
sound 1500 duration 500 wait
```

This statement would play a note of 1500 Hz for half a second. It would then move on to the next statement in the program.

## Delay

The delay statement is frequently used in programs to, er, delay them. The delay statement is followed by the size of the delay, **in tenths of a second**.

```
delay 5
```

The above statement would cause a robot program to delay for half a second. Note that any robot movement or sound playback would continue.

## Printing values

Your programs can print information back to the serial connection on the Arduino. Of course, printed messages will only mean anything if something is actually connected to the serial port. If you use the Python terminal program you will see your printed messages displayed in the output window.

### Print a single value

The print command will print a single value to the serial port.

```
print "hello"
```

The text to be printed is enclosed in double quotes, as shown above. If you print something else it will be appended to the line that is being printed.

A print statement can also print the value of an expression. You'll find out what these are later.

### Print and take a new line

If you want to print a statement and take a new line after it you can use the `println` command:

```
println "This is followed by a new line"
```

The println statement works in exactly the same way as print, except that it prints a new line after it has finished printing.

## Values and variables

A HullOS program can store and manipulate the value of variables. There is room in the Arduino for 20 variables. Each variable can hold a singed integer value. A variable has a name which is expressed as a letter, followed by a sequence of letters and numbers. You create a variable and set a value to it at the same time:

```
go=50
```

This would create a variable called go which holds the value 50. Variables can be used everywhere we have seen a number in previous examples. In other words:

```
move go
```

- would move the robot 50 mm.

Variables can be used in simple expressions:

```
go = go + 10
```

This would increase the value in go by 10. Note that expressions must be **very** simple and take the form of two *operands* (in the case above these would be the values of go and 10) and one *operator* (in the case above this would be the operator +). If you want to do more complicated things with values you'll have to spread the calculation over several statements.

The following arithmetic operators are available.

| | |
|---|---|
| + | plus (addition) |
| - | minus (subtraction) |
| * | times (multiplication) |
| / | divide (integer division) |
| % | modulus (remainder after integer division) |

### System variables

HullOS provides a number of system variables that give access to values returned by the robot sensors. A system variable has a name which begins with the @ character.

#### Read the distance sensor value

The `@distance` variable returns the current reading of the distance sensor as an integer value in mm. You can use it anywhere you can use a variable, although you can't assign values to it. That would be silly.

```
println @distance
```

This statement would print the current reading.

#### Detect motor movement

The `@moving` variable returns 1 if the motors are moving and 0 if they are not. You can use this to determine whether or not a given movement accept has been accepted.

```
println @moving
```

This statement would print 1 if the motors are moving and 0 if they are not.

#### Get random numbers

The `@random` variable returns a random number between 1 and 12. You can use this to give your robot random behaviours.

```
move @random*10
```

This statement make the robot move a random distance between 10 and 120 mm.

## HullOS Scripting

Individual commands can be sent to robots at any time, but you can also enter a program into a device controlled by HullOS. HullOS programs are expressed as a series of statements. Each statement must be given on a single line. A HullOS script is enclosed in the keywords `begin` and `end`. Programs can contain comments which are expressed using the # character at the start of the line:

```
begin
# This is an empty program
end
```

The case of a keyword is ignored.

```
BEGIN
# This is a shouty empty program
END
```

### Conditional Statements

A program can make decisions using an if construction. The if construction is followed by a Boolean expression which can be either true or false. If the condition is true the statement following the if condition is performed. The conditional statement can be followed by an else element which is obeyed if the statement is false

```
begin
if @distance < 100
  red
else
  green
end
```

The program above would turn the pixel red if the distance sensor was reading a distance of less than 100, otherwise the pixel would be turned green.

The language uses "Python style" indenting to indicate which statements are controlled by a condition.

```
begin
if @distance < 100
  sound 2000
  red
else
  sound 1000
  green
end
```

In the statements above the sound and the colour selection statements would be controlled by conditions. Conditions can be nested inside each other.

```
begin
silent=1
if @distance < 100
  if silent==0
    sound 2000
  red
else
  if silent==0
    sound 1000
  green
end
```

This program uses a variable called `silent` to turn the sounds off. If the value of `silent` is 1 the sound is not played.

You don't need to provide the else if your program doesn't need it.

The following logical conditions are available.

| | |
|---|---|
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| == | equal to |
| != | not equal to |

As with arithmetic expressions, a logical expression can only be comprised of two operands and one logical operator.

## Indent statements in conditions

The indenting for a given block level **must** be consistent within that block. In other words, if you indent the first statement after the if with two extra spaces, you must indent all the statements controlled by that if with two extra spaces as well. This means that the following statement is invalid:

```
begin
if @distance < 100
  sound 2000
   red
else
  sound 1000
  green
end
```

The highlighted red statement is indented too far. This will cause the program to be rejected by the robot and an error to be displayed.

```
Line:  4 Error: 34    red

RA

Errors
```

The error gives the line number and the number of the error that was detected. You can find a table of error numbers and their explanation at the end of this document.

You also use indenting when creating loops, as in the next section.

## Creating loops

A loop is a piece of code that is repeated multiple times. There are two looping constructions, `forever` and `while`.

### Loop code forever

A loop which continues forever will never stop. The `forever` statement is followed by the block of statements to be repeated indefinitely.

```
begin
forever
  red
  delay 5
  blue
  delay 5
end
```

The above program would flash the pixel red and blue, with a half second delay between each flash. Note that you can place statements after the loop, but these will never be obeyed.

### Loop on condition

The while statement is used to repeat a block of statements while a condition is true:

```
begin
while @distance>99
  green
magenta
end
```

The above program would display green until the distance value falls below 101. The it would display magenta.

### Break out of loops

A program can use the `break` statement to break out of a loop. When a program breaks out of a loop the program continues running at the statement following the last statement of the loop.

```
begin
green
forever
  if @distance<100
    break
red
end
```

The above program uses the break statement to break out of the forever loop when the distance sensor reading drops below 100. You can break out of forever or while loops. A break will only break out of one enclosing loop.

### Continue loops

The continue loop will cause a loop to return to the top of the loop and continue from there. If the loop is a while loop the loop condition will be tested.

```
begin
forever
  if @distance<100
    red
    continue
  sound 1000
  blue
red
end
```

The continue statement will cause the loop to be restarted, meaning that the sound is only made when the distance value is less than 100.

## HullOS Star commands

### Statement format

The star (*) commands provide the underlying behaviours of the robot. Each star command is expressed as a leading * character followed by a two-character command. The first character identifies the *family* of the command and the second character identifies the *function*.  A command is followed by one or more parameters, which are separated by commas. A parameter can be a numeric value or an expression in the same form as scripting expressions.

# Information: Initial character I

## Version: *IV

```
*IV
```

The robot responds with the version of the software followed by a linefeed character. The version takes the form of the message:

```
HullOS Version cn.nn
```

## Distance: *ID

```
*ID
```

The robot responds with current reading from the distance sensor followed by a linefeed character.

## Program: *IP

```
*IP
```

The robot responds with a listing of the current program.

## Status: *IS

```
*IS
```

The robot responds with current program status as a single digit value:

```
0       PROGRAM_STOPPED
1       PROGRAM_PAUSED
2       PROGRAM_ACTIVE
3       PROGRAM_AWAITING_MOVE_COMPLETION
4       PROGRAM_AWAITING_DELAY_COMPLETION5
5       SYSTEM_CONFIGURATION_CONNECTION
```

A status value of 5 indicates that the serial connection is actually to a configuration program rather than a live robot. This is used during configuration of the network processor in the Hull Pixelbot. You will never see this status value during normal operation. The configuration options for the robot are described in the document "Hull Pixelbot Configuration".

## Messaging: *IM

```
*IMlevel
```

This command is used to set the level of messaging that is produced by the robot when programs are running. The command is followed by a decimal value that sets the messaging level. The message levels are set as bits in the messaging level value.

| | |
|---|---|
| STATEMENT_CONFIRMATION 1 | outputs a confirmation message after each statement.<br>This will either be xxOK, where xx is the command, or xxFAIL: reason. |

| | |
|---|---|
| | Some commands, for example conditional jumps, will give additional information. |
| `LINE_NUMBERS 2` | displays the program position of each statement prior to execution |
| `ECHO_DOWNLOADS 4` | echoes each downloaded statement during a remote download |
| `DUMP_DOWNLOADS 8` | dumps a downloaded program before executing it |

If the corresponding bitfield is set the program will output information as described. Note that these commands can result in significant traffic on the serial connection and are only intended to be used for debugging.

When the robot is restarted the message level is set to 0 (i.e. all messages are turned off).

## Movement: Initial character M

### Forwards: *MF

```
MFdistance,time
```

The robot moves the number of steps given by the decimal value `distance`. If the number is negative the robot moves backwards that number of steps. If the robot is already moving this command will replace the existing one. The command can be followed by a comma and an optional `time` value that give the number of ticks (tenths of a second) that the move will take to complete. If the time value (and the comma) are omitted the robot will move as fast as possible.

```
*MF100
```

This would move the robot forwards 100 mm as quickly as possible.

```
*MF150,100
```

This would move the robot 150 mm and take 10 seconds to complete (remember that a “tick” is a tenth of a second).

The robot starts moving as soon as the command is received. If the move can be performed and the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
*MFOK
```

Note that this does not meant that the move command has been completed, rather that the robot has received and understood the command and has started moving.

If the time requested is not possible because the robot cannot move that quickly (for example move 100 mm in 1 tick) the move will not take place. If the `STATEMENT_CONFIRMATION` flag is set the robot replies with an error message:

```
*MFFail
```

### Rotate: MR

```
*MRangle,time
```

The robot rotates clockwise the given number of degrees given by the value `angle` (there are 360 degrees in a circle). If the number is negative the robot rotates anticlockwise that number of steps. If the robot is already moving this command will replace the existing one. If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
*MROK
```

Note that this does not meant that the move command has been completed, rather that the robot has received and understood the command and has started moving.

If the time is omitted the robot will perform the rotation as quickly as possible.

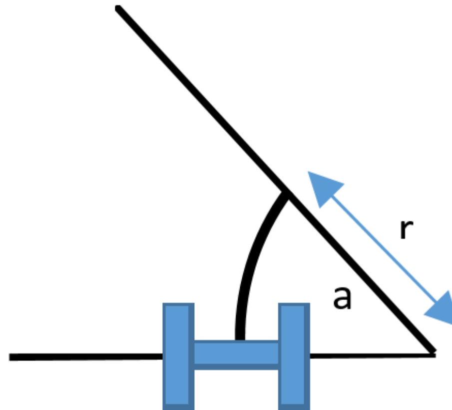
If the time requested is not possible because the robot cannot rotate that quickly (for example rotate 200 mm in 1 tick) the move will not take place. If the `STATEMENT_CONFIRMATION` flag is set the robot replies with an error message:

```
*MRFail
```

## Move Arc: *MA

```
*MAradius,angle,time
```

This statement allows the robot to move in an arc. The first parameter is the `radius` of the arc, the second is the `angle` giving the angular distance around the arc and the third is the `time` over which the move is to take place.




The figure above shows how this works. If the radius is positive the curve will be about a point which is to the right of the robot, and the robot will turn clockwise as it moves. If the radius is negative the curve will be about a point that is to the left of the robot, and the robot will turn counterclockwise. The centre point of the arc is on a line drawn between the two wheels. If the angle is positive the robot will turn clockwise.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
MAOK
```

Note that this does not meant that the move command has been completed, rather that the robot has received and understood the command and has started moving.

If the time requested is not possible because the robot cannot move that quickly the move will not take place. If the `STATEMENT_CONFIRMATION` flag is set the robot replies with an error message:

```
MAFail
```

As an example the following statement would cause the robot to traverse a complete circle and take a minute to perform this:

```
*MA100,360,600
```

The radius of the circle is 100mm, the distance around the circle is 360 degrees and the move will take 600 ticks (remember that a tick is a $10^{th}$ of a second)

If the value of the radius is given as 0 the robot will rotate about its centre:

```
*MA0,90,600
```

This would cause the robot to rotate 90 degrees about its centre over 600 ticks.

## Move Motors: *MM

```
*MMleft,right,time
```

This statement provides direct control of the speed and distance moved of each individual wheel. It is followed by three integer values that give the distance to be moved for each motor and the time to be taken for the move.

When the motor has moved the specified distance it stops. The distance values can be signed, in which case the motor will run in reverse.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
MMOK
```

Note that this does not meant that the move command has been completed, rather that the robot has received and understood the command and has started moving.  If the turn could not be performed a fail message is generated followed by a value which indicates the reason for the failure.

```
MMFail: d
```

The value of d gives the reason why the move could not take place:

```
1 Left_Distance_Too_Large
2 Right_Distance_Too_Large
3 Left_And_Right_Distance_Too_Large
```

### Check moving: *MC

```
*MC
```

This command can be used to determine whether the robot has completed a requested move operation. If the motors are still moving the robot replies with:

```
MCmove
```

If the motors are stopped the robot replies with:

```
MCstopped
```

Note that these messages are sent irrespective of the state of the `STATEMENT_CONFIRMATION` flag.

### Stop robot: *MS

```
*MS
```

Stops any current move behaviour.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
MSOK
```

### Wheel configure: *MW

```
*MWlll,rrr,sss
```

lll  left wheel diameter in mm
rrr  right wheel diameter in mm
sss  wheel separation in mm

This command can be used to configure the dimensions and spacing of the wheels fitted to the robot. The values are used to calculate all the robot movement. The values are stored in EEPROM inside the robot. The very first time the robot is turned on the dimensions are set to their default values:

Wheel diameter  69mm
Wheel spacing  110mm

These are the dimensions based on the stl files for the robot.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
MWOK
```

Note that these values are not validated in any way, so if you put silly dimensions you may find that silly things happen. And quite right too.

## View wheel configuration: *MV

```
*MV
```

This allows you to view the current settings of the wheel configuration values. The display is as follows:

```
Wheel settings
Left diameter: 69
Right diameter: 69
Wheel spacing: 110
```

## Pixel control: Initial character P

The Hull Pixelbot can be fitted with a coloured “pixel”. This can be a single pixel, or it can be composed of a ring of 12 pixels. The Pixel control commands allow you to control individual pixels in the ring, or set an overall “flickering” colour. The rate at which the colour flickers can also be controlled.

### Remote Named Colour Candle: *PN

```
*PNc
```

Sets the pixel display to show a flickering candle of the given colour. This sets the colour of all the pixels in the display. The required colour is specified by a single character:

- R – red
- G – green
- B – blue
- Y – yellow
- M - magenta
- C – cyan
- W – white
- K - black

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PCNOK
```

If any of the values are missing an appropriate message is displayed, for example:

```
PC
```

- would generate the error:

```
FAIL: mising colour in set colour by name
```

Note that this message is only output if the STATEMENT_CONFIRMATION flag is set. If the colour selection character is invalid the following message is displayed.

```
"FAIL: invalid colour in set colour by name
```

Note that these messages are only output if the STATEMENT_CONFIRMATION flag is set.

### Remote Coloured Candle: *PC

```
*PCrrr,ggg,bbb
```

| | |
|---|---|
| rrr | red intensity in range 0-255 |
| ggg | green intensity in range 0-255 |
| bbb | blue intensity in range 0-255 |

Sets the pixel display to show a flickering candle of the given colour. This sets the colour of all the pixels in the display.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PCOK
```

If any of the values are missing an appropriate message is displayed, for example:

```
PC255,
```

- would generate the error:

```
PCFAIL: mising colours after red in readColor
```

Note that this message is only output if the STATEMENT_CONFIRMATION flag is set.

### Pixels Off: *PO

```
*PO
```

Turns off all the pixels in the pixel display.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
POOK
```

### Pixels Flicker update speed: *PF

```
*PFnn
```

Sets the flicker update speed for the pixels. The larger the number, the faster the pixels will change colour. The speed is given in the range 1 to 20. A speed of 1 is very gentle, a speed of 20 is manic. When the program starts the speed is set to 8. Values outside the range 1-20 are clamped.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PFOK
```

### Remote Set Individual Pixel: *PI

```
*PIppp,rrr,ggg,bbb
```

ppp number of the pixel to be set in the range 0 to n-1, where n is the number of pixels

rrr red intensity in range 0-255
ggg green intensity in range 0-255
bbb blue intensity in range 0-255

Set Individual to the given colour. The state and colours of the other pixels are not affected by this.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PIOK
```

If any errors are detected in the values supplied an appropriate message is displayed instead of the OK message.

### Remote Crossfade colour: *PX

```
*PXss,rrr,ggg,bbb
```

ss fade speed in range 1-20
rrr red intensity in range 0-255
ggg green intensity in range 0-255
bbb blue intensity in range 0-255

Sets the pixel display to cross fade to a flickering candle of the given colour. This sets the colour of all the pixels in the display.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PXOK
```

If any of the values are missing an appropriate message is displayed, for example:

```
PX20,
```

- would generate the error:

```
PXFAIL: mising colours after red in readColor
```

Note that this message is only output if the STATEMENT_CONFIRMATION flag is set.

### Pixel Animate: *PA

```
*PA
```

Turns on pixel animation so that the pixels flicker. This is the default setting.

### Pixel Stationary: *PS

```
*PS
```

Turns of pixel animation. The pixels no-longer flicker, although colour selections will still be implemented. This command is only necessary if you are playing sounds and you don’t want the sound output to be slightly distorted by the neopixel commands that are being repeatedly sent to implement the flicker. If the pixels are stationary any crossfade that has been specified will not take place.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PSOK
```

### Pixels Random: *PR

```
*PR
```

Sets all the pixels to different, random, colours flickering at random rates.

If the STATEMENT_CONFIRMATION flag is set the robot replies with:

```
PROK
```

## Sound Output: Initial character S

These commands can be used to generate sounds using the speaker fitted to the Hull Pixelbot. The sound is a simple square wave. Sounds can be played while the robot is moving and the lights are flickering, but there is some interruption of the sound output by the process of sending commands to the lights. If you want to generate pure tones you can use the PS command to disable flickering.

### Sound tone: *ST

```
*STfreq,duration,wait
```

The robot produces a sound output of the given frequency and duration. The frequency is given in Hz and the duration in milliseconds. The sound starts when the command is performed. If the wait element is W the program pauses while the sound plays. If the wait element is N the program continues onto the next statement, with the sound playing in the background.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
SOUNDOK
```

If any of the values are missing an appropriate message is displayed, for example:

```
ST
```

- would generate the error:

```
FAIL: no tone frequency
```

Note that this message is only output if the `STATEMENT_CONFIRMATION` flag is set.

As an example, the command:

```
ST1000,500,W
```

- would play a 1000 hz tone for half a second. The program would pause while the sound is played.

# Program Control: Initial character C

These commands are performed when the robot is running a program sequence.  The can be entered directly into the robot, but they won't do much.

## Pause when the motors are active: *CA

    *CA

The program pauses while the motors are active. This allows a program to wait until a movement has completed. If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

    CAOK

Note the reply is sent when the command has been received **not** when the robot has completed the pause.

## Delay: *CD

    *CDddd

The program pauses for the number of *ticks* given by the decimal value ddd. The number of ticks can be omitted:

    CD

The program repeats the previous delay. If there was no previous delay the program does not delay. The delay starts as soon as the command is received.

A tick is a tenth of a second.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

    CDOK

Note the reply is sent when the command has been received **not** when the robot has completed the delay. Ongoing move commands will continue to complete, and the robot will respond to other direct commands.

## Label: *CL

    *CLcccc

This statement declares a label which can be used as the destination of a jump instruction. The label can be any number of characters and will be terminated by the end of the statement.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

    CLOK

## Jump: *CJ

    *CJcccc

This statement causes execution of the program to continue from the given label. The label can be any number of characters and will be terminated by the end of the statement. The program must contain the label requested, or the program will stop.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

If the label is found and the jump performed the robot replies:

    CJOK

If the label is missing from the program the robot replies with:

```
CJFAIL: no dest
```

### Coin toss jump: *CC

```
*CCcccc
```

This statement causes execution of the program to continue from the given label fifty percent of the time. The rest of the time the program continues. The label can be any number of characters and will be terminated by the end of the statement. The program must contain the label requested, or the program will stop.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

If the label is found and the jump performed the robot replies:

```
CCjump
```

If the label is found and the jump not performed the robot replies:

```
CCcontinue
```

If the label is missing from the program the robot replies with:

```
CCFAIL: no dest
```

### Jump when motors inactive: *CI

```
*CIccc
```

The program will jump to the given label if the motors are inactive.  The robot replies with:

```
*CIOK
```

### Measure Distance: *CM

```
*CMddd,cccc
```

This statement causes execution of the program to continue from the given label if the distance sensor reading is less than the given distance value. The value is given in centimetres.

The label can be any number of characters and will be terminated by the end of the statement. The program must contain the label requested, or the program will stop. If the distance measured is greater than the given value, the program continues at the next statement.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

If the label is not present in the instruction the robot replies with:

```
CMFail: missing dest
```

If the label is found the robot replies with:

```
CMFail: label not found
```

If the label is found and the distance is less the robot replies with:

```
CMjump
```

If the label is found and the distance is greater the robot replies with:

```
CMcontinue
```

### Compare Condition: *CC

```
*CCeeee,cccc
```

This statement causes execution of the program to continue from the given label if the given expression to True.

The label can be any number of characters and will be terminated by the end of the statement. The program must contain the label requested, or the program will stop. If the given expression is false the program continues at the next statement.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

If the label is not present in the instruction the robot replies with:

```
CCFail: missing dest
```

If the label is found the robot replies with:

```
CCFail: label not found
```

If the label is found and the distance is less the robot replies with:

```
CCjump
```

If the label is found and the distance is greater the robot replies with:

```
CCcontinue
```

### Compare Condition not true: *CN

```
*CFeeee,cccc
```

This statement causes execution of the program to continue from the given label if the given expression evaluates to True.

The label can be any number of characters and will be terminated by the end of the statement. The program must contain the label requested, or the program will stop. If the given expression is false the program continues at the next statement.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

If the label is not present in the instruction the robot replies with:

```
CCFail: missing dest
```

If the label is found the robot replies with:

```
CCFail: label not found
```

If the label is found and the distance is less the robot replies with:

```
CCjump
```

If the label is found and the distance is greater the robot replies with:

```
CCcontinue
```

## Remote Management: Initial character R

These commands are performed to allow a new stored program to be downloaded into the EEPROM in the robot and to control program execution by the robot drive system.

### Start Program: *RS

This statement starts the execution of the current program (if present).

```
*RS
```

This statement is obeyed immediately upon receipt.  The program is started from the first statement in the program.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
RSOK
```

### Halt Program: *RH

This statement halts the execution of the current program (if present).

```
*RH
```

This statement is obeyed immediately upon receipt. The program is halted. It cannot be resumed, it must be restarted.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
RHOK
```

### Pause Program: *RP

This statement pauses the execution of the current program (if present).

```
*RP
```

This statement is obeyed immediately upon receipt. The program is paused. It can be resumed using the RR statement.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
RPOK
```

### Resume Running: *RR

This statement resumes the execution of the current program (if it has been paused).

```
*RR
```

This statement is obeyed immediately upon receipt. The program is resumed.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following messages:

```
RROK
```

This is displayed if the program has been resumed.

```
RRFail:d
```

This is displayed if the program cannot be resumed. The single digit value following the Fail: gives the current program state as for the IS command.

### Remote Clear: *RC

This statement clears the current program from EEPROM.

```
*RC
```

This statement is obeyed immediately upon receipt. The program is stopped if it is running.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with:

```
RCOK
```

### Remote Download: *RM

This statement stops the execution of the current program and switches off the pixel display ready for the receipt of a new program.

```
*RM
```

The program is sent as a sequence of statements that directly follow the RM command. Each statement is stored in EEPROM for execution by the robot when the download is complete. A standard Arduino has enough internal storage for around 900 bytes of remote program storage.

Note that the RM command is terminated by a Carriage Return (CR) character (0x0D) as are all commands. Each statement in the program is separated from the next by the CR character. The end of the program is specified by an RX command (see next).

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following message:

```
RMOK
```

If the ECHO_DOWNLOADS flag is set the robot echoes each statement as it is received.

### Remote Download Exit: *RX

This statement completes a download started by the *RM command.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following message:

```
RXOK
```

The program is executed immediately.

If the DUMP_DOWNLOADS flag is set the robot dumps a listing of the received program as soon as it has been successfully received.

### Remote Download Abandon: *RA

This statement abandons a download started by the *RM command. The robot returns to immediate mode, i.e. commands are performed as soon as they are received.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following message:

```
RAOK
```

## Variable Management: Initial character V

These commands are performed to allow a program to manipulate variables. Each variable is identified by a name that must start with a letter and contain only letters and digits. Variable name length is limited to six characters and there are a limited number of variables available. Variables are stored and manipulated as integers.

A variable must be assigned before the value in it is used.

### Clear Variables: *VC

This statement clears all the variables from memory. It is performed when a program starts running.

```
*VC
```

This statement is obeyed immediately upon receipt.

If the STATEMENT_CONFIRMATION flag is set the robot replies with the following message:

```
VCOK
```

### Set Variable: *VSx=e

This statement sets the value of a variable to the result of an expression. The expression can contain two operators and a single operator which can be +, -, * or /.

```
*VScount=0
*VScount=count+1
*VSresult=count*5
```

This statement is obeyed immediately upon receipt. If the variable does not exist it is created automatically.

If the STATEMENT_CONFIRMATION flag is set the robot replies with the following messages:

```
VSOK
VS no equals after variable name
VS no room for variable
```

### View Variable: *VVx

This statement displays the value held in the specified variable. If the variable does not exist an error is produced which is displayed if the STATEMENT_CONFIRMATION flag is set.

```
VV variable not found
```

### System values

An expression can contain literal values (integers), variables and system values. A system value is pre-ceded by the @ character and allows a program access to system information. The following values are available:

#### *@distance*

This is the current reading of the distance sensor, in centimetres.

#### *@light*

This is the current reading of the light sensor, which is connected to A0. The value is in the range 0-1023

#### *@moving*

This value is 1 if the motors are moving, 0 if not.

#### *@random*

This is a random integer in the range 1-12 inclusive.

## W – write initial character w

These commands are performed to write the contents of a variable or a string of text to the serial console.

### Write Text: *WT

This statement writes text to the serial terminal. The text follows the command immediately. The end of the line marks the end of the text to be written. The statement does not take a newline. The command WL can be used to take a newline.

```
*WTHello world
```

This statement is obeyed immediately upon receipt. There is no confirmation message sent when the statement is performed. The above statement would output "Hello world" to the serial console. Note that trailing or leading spaces are not trimmed from the text to be written.

### Write Line: *WL

This statement writes a newline to the serial terminal.

```
*WL
```

### Write Value: WV

This statement writes an expression to the serial terminal.

```
*WVexpr
```

The expression is evaluated and then displayed as an integer value.

If the `STATEMENT_CONFIRMATION` flag is set the robot replies with the following message:

```
VCOK
```

## N – network configuration initial character N

These commands are performed to configure the Hull Pixelbot Network client.

### Network Read: *NR

Read the settings from the network client as a sequence of strings:

```
*NR
```

The network client response with a sequence of lines separated by the linefeed character. Note that the passwords and the Azure key are replaced with the sequence "******" to indicate that the password has been set.

```
wifi1
password1
wifi2
password2
wifi3
password3
wifi4
password4
wifi5
password5
Robot ID
Azure endpoint
Azure Key
```

### Network Store: *NS

Store the settings from the network client as a sequence of strings:

```
*NS
```

The host must send a set of character strings that contain the configuration information.

```
wifi1
password1
wifi2
password2
wifi3
password3
wifi4
password4
wifi5
password5
Robot ID
Azure endpoint
Azure Key
```

If the password or Azure Key information is supplied as the sequence "******" this means that the password in the robot will not be changed for that field.

### Access point list: *NA

Lists the access points presently visible to the network client:

```
*NA
```

The network client responds with a list of the access points that have been discovered in the following format:

```
Listing access points

1 network(s) found

1: ZyXEL56E8A7, Ch:11 (-75dBm)
```

### Network connect: NC

Connects the network client to the Azure server.

```
*NC
```

The network connection

## HullOS Editor

The HullOS Editor is a Python program that you can use to create programs and send them to the robot.

### Installing the HullOS Editor

You can install the HullOS editor on any machine running Python 3.6 or greater. It requires the pyserial library. You can download it from here:

https://github.com/HullPixelbot/HullOSEditor

There is also a manual available at the same location.

## Error Messages

These are the error messages for the scripting language and a brief explanation of what they mean:

```
#define ERROR_OK 0
#define ERROR_MISSING_TIME_VALUE_IN_INTIME 1
#define ERROR_INVALID_INTIME 2
#define ERROR_MISSING_MOVE_DISTANCE 3
#define ERROR_NOT_IMPLEMENTED 4
#define ERROR_MISSING_ANGLE_IN_TURN 5
#define ERROR_INVAILD_ANGLE_SEPARATOR_IN_ARC 6
#define VARIABLE_USED_BEFORE_IT_WAS_CREATED 7
#define ERROR_INVALID_DIGIT_IN_NUMBER 8
#define ERROR_INVALID_HARDWARE_READ_DEVICE 9
#define ERROR_INVALID_VARIABLE_NAME_IN_SET 10
#define ERROR_TOO_MANY_VARIABLES 11
#define ERROR_NO_EQUALS_IN_SET 12
#define ERROR_CHARACTERS_ON_THE_END_OF_THE_LINE_AFTER_WAIT 13
#define ERROR_INVALID_OPERATOR_IN_EXPRESSION 14
#define ERROR_MISSING_TIME_IN_DELAY 15
#define ERROR_MISSING_RED_VALUE_IN_COLOUR 16
#define ERROR_MISSING_GREEN_VALUE_IN_COLOUR 17
#define ERROR_MISSING_BLUE_VALUE_IN_COLOUR 18
#define ERROR_MISSING_OPERATOR_IN_COMPARE 19
#define ERROR_ENDIF_BEFORE_IF 20
#define ERROR_INVALID_CONSTRUCTION_MATCHING_ENDIF 21
#define FOREVER_BEFORE_DO 22
#define ERROR_INVALID_CONSTRUCTION_MATCHING_FOREVER 23
#define UNTIL_BEFORE_DO 24
#define ERROR_INVALID_CONSTRUCTION_MATCHING_UNTIL 25
#define ERROR_INVALID_COMMAND 26
#define ERROR_MISSING_OPERATOR_IN_WHILE 27
#define ERROR_INVALID_CONSTRUCTION_MATCHING_ENDWHILE 28
#define ENDWHILE_BEFORE_WHILE 29
#define ERROR_SCRIPT_INPUT_BUFFER_OVERFLOW 30
#define ERROR_INTIME_EXPECTED_BUT_NOT_SUPPLIED 31
```

```
#define ERROR_ELSE_WITHOUT_IF 32
#define ERROR_INDENT_OUTWARDS_WITHOUT_ENCLOSING_COMMAND 33
#define ERROR_INDENT_INWARDS_WITHOUT_A_VALID_ENCLOSING_STATEMENT 34
#define ERROR_INDENT_OUTWARDS_HAS_INVALID_OPERATION_ON_STACK 35
#define ERROR_INDENT_OUTWARDS_DOES_NOT_MATCH_ENCLOSING_STATEMENT_INDENT 36
#define ERROR_MISSING_CLOSE_QUOTE_ON_PRINT 37
#define ERROR_MISSING_PITCH_VALUE_IN_SOUND 38
#define ERROR_MISSING_DURATION_VALUE_IN_SOUND 39
#define ERROR_NOT_A_WAIT_AT_THE_END_OF_SOUND 40
#define ERROR_BEGIN_WHEN_COMPILING_PROGRAM 41
#define ERROR_END_WHEN_NOT_COMPILING_PROGRAM 42
#define ERROR_STOP_WHEN_COMPILING_PROGRAM 43
#define ERROR_RUN_WHEN_COMPILING_PROGRAM 44
#define ERROR_CLEAR_WHEN_COMPILING_PROGRAM 45
#define ERROR_FOREVER_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 46
#define ERROR_WHILE_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 47
#define ERROR_IF_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 48
#define ERROR_ELSE_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 49
#define ERROR_ELSE_MUST_BE_PART_OF_AN_INDENTED_BLOCK 50
#define ERROR_NO_LABEL_FOR_LOOP_ON_STACK_IN_BREAK 51
#define ERROR_BREAK_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 52
#define ERROR_INVALID_COMMAND_AFTER_DURATION_SHOULD_BE_WAIT 53
#define ERROR_SECOND_COMMAND_IN_SOUND_IS_NOT_DURATION 54
#define ERROR_CONTINUE_CANNOT_BE_USED_OUTSIDE_A_PROGRAM 55
#define ERROR_NO_LABEL_FOR_LOOP_ON_STACK_IN_CONTINUE 56
#define ERROR_NO_RADIUS_IN_ARC 57
#define ERROR_NO_ANGLE_IN_ARC 58
```

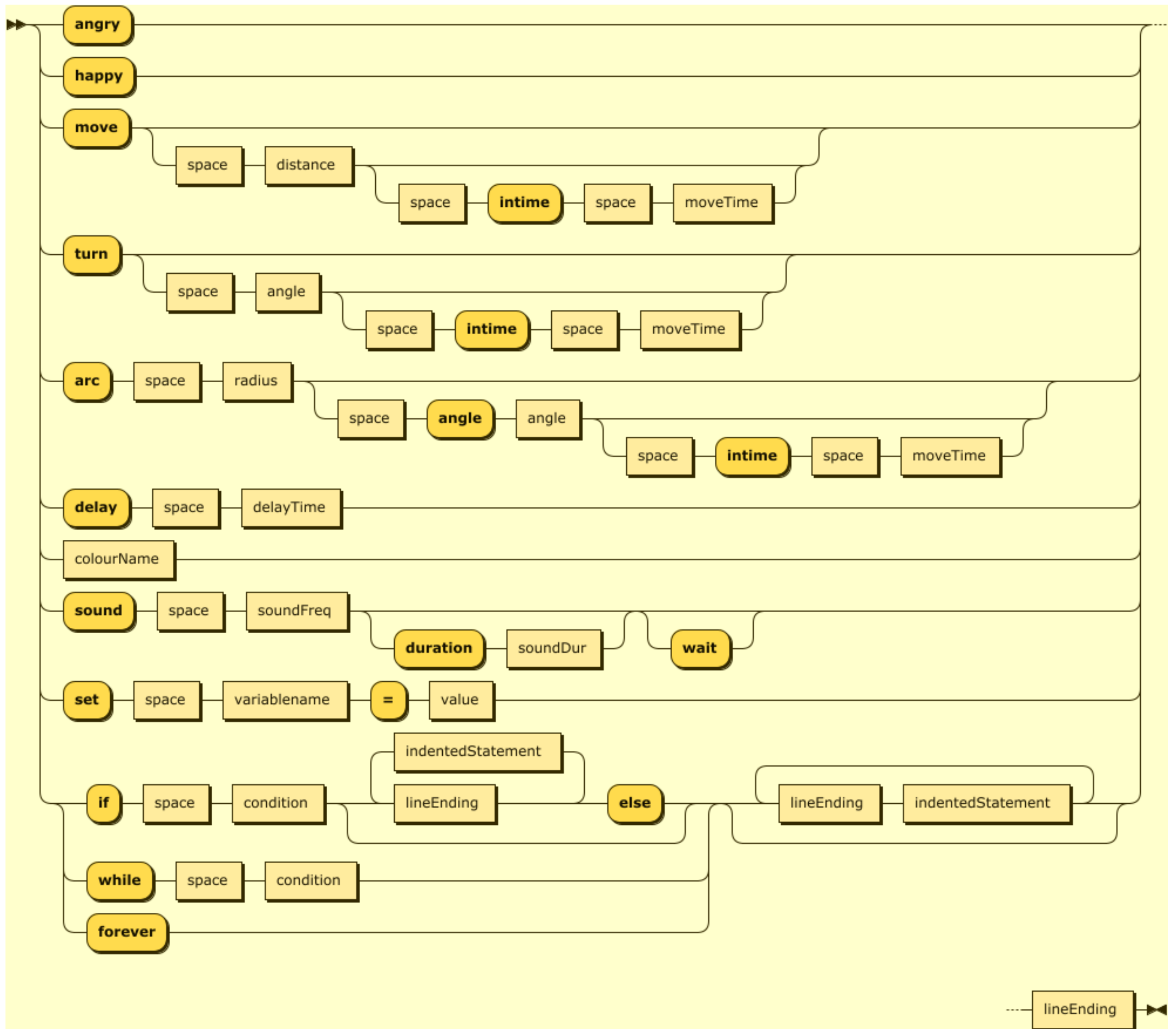
## Language Specification

The language can be specified by the following grammar:

```
letter ::= ('A' |'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' |'L' | 'M' | 'N' | 'O' | 'P'
| 'Q' | 'R' | 'S' | 'T' | 'U' |'V' | 'W' | 'X' | 'Y' | 'Z' )

digit ::= ( '1' |'2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0' )

integer ::= ( ('-'|'+')? digit*)

space ::= ( ' ' (' ')*)

indentedStatement ::= space statement

variablename ::= letter (letter | digit)*

readingName ::= '%' variableName

operand ::= variablename | integer | readingName

operator ::= '+' | '-' | '*' | '/' | '%'

expression ::= operand spaces operator spaces operand

value ::= (operand | expression)

logicalOperator ::= '<' | '>' | '==' | '!='

condition ::= (operand logicalOperator operand)

statement ::= ('angry' lineEnding)
statement ::= ('happy' lineEnding)

distance ::= (value)
moveTime ::= (value)

statement ::= ('move' (space  distance ( space 'intime' space moveTime)?)? lineEnding)

angle ::= (value)

statement ::= ('turn' (space angle ( space 'intime' space moveTime)?)? lineEnding)
```

```
radius ::= (value)

statement ::= ('arc' space radius (space 'angle' angle ( space 'intime' space moveTime )?)? lineEnding )

delayTime ::= (value)

statement ::= ('delay' space  delayTime lineEnding)

colourName ::= 'red' | 'green' | 'blue' | 'yellow' | 'magenta' | 'cyan' | 'white' | 'black'

statement ::= colourName lineEnding

soundFreq ::= (value)
soundDur ::= (value)

statement ::= (('sound') space soundFreq ('duration' soundDur)? ('wait')? lineEnding )

statement ::= ('set' space variablename '=' value lineEnding)

statement ::= ('if' space condition lineEnding (indentedStatement lineEnding)* (('else' lineEnding)
(indentedStatement lineEnding)*)?)

statement ::= ('while' space condition lineEnding (indentedStatement  lineEnding)*)

statement ::= ('forever' lineEnding (indentedStatement  lineEnding)*)

program ::= 'begin' (statement)* 'end'
```

This can be rendered into a “railroad” diagram that shows the language syntax.


angry
happy
move
space
distance
space
intime
space
moveTime
turn
space
angle
space
intime
space
moveTime
arc
space
radius
space
angle
angle
space
intime
space
moveTime
delay
space
delayTime
colourName
sound
space
soundFreq
duration
soundDur
wait
set
space
variablename
=
value
indentedStatement
if
space
condition
lineEnding
else
lineEnding
indentedStatement
while
space
condition
forever
lineEnding