User’s Guide to the HP DMF User Interface and Macro Language

Evan Kirshenbaum

*Thursday, October 14, 2021*

What this document is and what it is not

This document explains how to install and run the user interface software for interactive control of the prototype HP digital microfluidics board. It also covers the syntax and concepts for the DMF macro language that can be used for more complex control via the user interface.

Other documents will cover the installing and running the physical board as well as the Python API that allows the simple specification of complete protocols using the board.

Installing the software

This section assumes that you have a copy of the Python source code, probably obtained as a zip file.

Python version

The code has been tested with Python version 3.9.4. Versions less than 3.9 are known to not work.

Required packages

The following packages are required. They can be installed using **pip**:

* antlr4-python3-runtime
* clipboard
* matplotlib
* pyserial

Setting $PYPATH

The following directories must be included in your $PYTHONPATH environment variable:

* ${ROOT}/src
* ${ROOT}/tools
* ${ROOT}/target/generated-sources/antlr4

Depending on how the zip file was generated ${ROOT} will either be the root directory of the zip file or a subdirectory called mpam.

Running the software

This section assumes that you are trying to control a “Wombat” board (an HP DMF board controlled via an OpenDrop controller). In this configuration only the following are available (assuming a board orientation with the OpenDrop controller on the top):

* The bottom two wells on each side. On the full boards, the wells are numbered from 0 to 3 down the left side and 4 to 7 down the right side. Thus, wells 2, 3, 6, and 7 are available.
* The three pads leading away from (in the same rows as the exit pads for) wells 2, 3, and 7.
* The two pads leading away from (in the same row as the exit pad for) well 6.
* All of the pads in rows 1 through 5 (with zero being the bottom row).

The UI is designed to be run from the command line. The basic structure is

python wombat *task arguments…*

where the wombat program can be found in the tools directory. For simple interactive control, the task you want to run is called display-only, which can be abbreviated to display:[[1]](#footnote-1)

python wombat display arguments …

Command-line arguments

The tool takes several command-line arguments. A full set can be obtained by giving the --help argument:

python wombat display --help

Several of these arguments take time values, which must be specified as a number followed by a time unit. Recognized units include

|  |  |
| --- | --- |
| ns, nsec | nanoseconds |
| us, usec | microseconds |
| ms, msec | milliseconds |
| s, sec, secs, second, seconds | seconds |
| min, minute, minutes | minutes |
| hr, hour, hours | hours |
| day, days | days |

Note that a time value is a single argument, so if a space is included between the number and the unit, the argument must be enclosed in quotes or the space must be escaped using a backslash.

The arguments taken by the program include the following.[[2]](#footnote-2) Note that long arguments (which start with two dashes) can be abbreviated to any unique prefix.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Long | Short | Type | Default | Description |
| --help |  |  |  | Print the help on the task and exit |
| --clock-speed *T* | -cs *T* | Time | 100 ms | The amount of time between clock ticks |
| --paused |  |  | false | Don’t start the clock |
| --initial-delay *T* |  | Time | 5 sec | The amount of time to wait before running the task. |
| --min-time *T* |  | Time | 5 min | The minimum amount of time to leave the display on, even if the operation has finished. |
| --max-time *T* |  | Time | no limit | The maximum amount of time to leave the display up. |
| --no-display | -nd |  |  | Run the task without the on-screen display |
| --update-interval *T* |  | Time | 20 ms | The maximum amount of time between display updates |
| --macro-file |  | File |  | A file containing DMF macro definitions |

For the wombat exerciser, there is also

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Long | Short | Type | Default | Description |
| --port *P* | -p *P* | String |  | The USB port to use to talk to the board. |

For interactive control, you will probably want to set --min-time to a largish value (e.g., 1 day) to prevent the program from exiting on you unexpectedly. (The display-only task takes no time to execute.) You can adjust the clock speed and whether the clock is paused from the display, so you need not specify it on the command line.

If you are planning on using macro language expressions, you will probably want to specify a macro file containing macros that you may want to make use of. The macro language will be described in detail below and an example file can be found in inputs/macros.dmf.

Finding the USB address

To control an actual Wombat board, the --port argument must be provided. (If it is not, the display will come up and you can interact with it, but it will have no effect on the physical board.)

To determine the USB address, the easiest way is to run a tool called USBDeview, which can be obtained at <https://www.nirsoft.net/utils/usb_devices_view.html>. The download link (near the bottom of the page) will download a zip file, and the program can be run directly from this file (without extracting it). Running it will bring up a display like

Graphical user interface, application, table

Description automatically generated

The device name you want to look for for the OpenDrop controller is “Feather M0”, and the option value is found in the “Drive Letter” column, in this case COM3, so the argument to specify would be --port COM3.

Command-line summary

In summary, the most likely command line to specify for pure interactive use will be something like

python tools/wombat display-only

--port COM3

--min-time 1day

--macro-file inputs/macros.dmf

(on a single line, of course).

The display

When you run the program, you should see a display like this:

Diagram

Description automatically generated

At the top of the display is a visual representation of the Wombat board with the parts of the board that are not controllable blacked out. Colored circles over pads represent drops, and the color indicates the presumed reagent (or mixture of reagents). The colors for reagents currently on the board are shown in the “Reagents” legend at the top right. (For interactive control, the only reagent you are likely to see is “unknown”.) The area of the circle is proportional to the size of the drop.

Note that drops shown on the board are inferred from mouse clicks and statements in the macro language—they do not necessarily correspond to actual physical drops on the board, as there is no actual drop sensing capability in the current version of the hardware.

Pads whose electrode is powered are shown with a green border. Those whose electrode is off are shown with a black border. When controlling the board via mouse clicks (described below), a thick border indicates the electrode state that will be transitioned to in the next clock tick (necessarily the opposite of its current state).

The numbers on some pads indicate the inferred temperature of thermal zones. These are inactive in the wombat board and should always report 22C (room temperature). The M on pad (2, 3) represents the state of a (similarly not yet controllable) magnet, and the small circle on pad (13, 3) represents an “extraction port” to be used for adding or removing fluid, typically under the control of a pipetting robot. All of these can be ignored at this point.

At the bottom of the display are buttons and text boxes that can be used to control the board. To the left are controls to pause (and resume) the clock, allow a paused clock to proceed for one clock tick, and to specify the clock speed in milliseconds. To the right is a text box used to enter macro language expressions.

Interactive control

Each of the sections of the display described above allows the user to interactively control the state of the physical board. On the board image, you can click on pads (including pads in wells) to turn them on and off. In the clock region, you can control the speed of the clock and whether or not it is running. And in the expression region you can issue commands written in the macro language to perform more complex actions.

Controlling the clock

The software maintains an internal clock that ticks at a steady rate.[[3]](#footnote-3) When multiple electrode state changes are specified at the same time, they all take place on the next clock tick. By default, the clock ticks once ever 100 ms, but it can be sped up or slowed down either by use of the –clock-speed command-line argument (e.g., --clock-speed 200ms) or by specifying a number in the text box.[[4]](#footnote-4)

The clock starts out running, but this can be changed by specifying the --pause command-line argument or by pressing the Pause button (which then turns into a Run button). When the clock is paused, multiple operations may be requested using the mouse or expression box, but they will not take effect until the Run button is pressed to restart the clock or the Step button is pressed to allow the clock to proceed for one tick and then reenter the paused state.

Using the mouse

You can click on the board to toggle the state of pads (including well pads). If the clock is running, the change of state happens on the next clock tick. If the clock is paused, the change of state will happen—well, still on the next clock tick, but this won’t happen until you press Run or Step. Until then, you will see thick borders around the pads scheduled to toggle, with the color indicating their new state. If you click a second time on a pad, it deselects it from the list of changes.

The effect of clicking was meant to be unsurprising, but it sounds complicated when described. Basically, the system tries to model the effect of what changing pad state means (ideally) to the drops on the board. In particular:

* If you hold down control while clicking, you specify that the pad should be toggled.
* If you don’t hold down control and you click on an empty pad next to a drop, it not only turns the clicked pad on but also schedules any neighboring pads that hold drops to turn off. The notion is that by not holding control, you are specifying that you want to “pull” the neighboring drop(s) toward the pad. Note that neighboring pads include diagonal neighbors.

The logic for what happens to modeled drops is as follows:

* If a pad is turned on and none of its neighbors hold a drop, a new drop is created at that pad. This can be useful to model physical drops added to the board that don’t match the model.
* If a pad is turned off and the drop isn’t pulled by a neighbor pad, the modelled drop is removed from the model. This can also be useful to correct errors.
* If a pad with a drop is turned off and pulled to a single neighbor, the entire drop is moved to that neighbor pad.
  + If the neighbor pad is a well gate pad, the drop is removed from the board and the volume of the well is increased.
  + If the pad is the exit pad from a well and the well’s gate was on, a new drop is created at the exit pad and the volume of the well is decreased.
* If a pad is turned off and pulled to multiple neighbors, the drop is split evenly and pulled in all of the directions.
  + If the drop is pulled toward a neighbor, but its own electrode is to remain on, it is similarly split.
* If multiple drops (or parts of drops) are moved to the same pad, they are merged into one larger drop at the destination pad.

Using macro language commands

Finally, in the text box at the bottom right, you can type macro language expressions. The next section will go into the language in detail, but some examples:

* d = drop @ (2,3)
  + Bind variable d to a drop at the pad at column 2, row 3 (with the bottom left corner being (0,0). If there is no drop there, one is created in the model.
* d : right 3
  + Move the drop d right three pads.
* d + 2 up : on
  + Turn on the pad two rows above drop d.
* d : right 2 : up 3
  + Move the drop two columns to the right and then 3 rows up.
* rectangle(d, 2 right, 3 up)
  + Call the macro bound to rectangle, which takes three arguments, a drop, and two “deltas” describing the directions and lengths of the first two sides.
* w = well #3
  + Bind variable w to well number three (the bottom well on the left-hand side).
* d = dispense(w)
  + Call the macro bound to dispense, passing in well w as an argument. Bind the resulting drop to d.
* d = w : dispense
  + The same things. The who : what “injection” syntax can be used for macros that take one argument.
* p = d + 1 up
  + Bind p to the pad immediately above d.
* [[ d : turn off; p : turn on; ]]
  + In parallel, turn off the pad under drop d and turn on the pad at p.

When you enter an expression (either by hitting ENTER or by clicking the Do it button), the expression will be printed to the console output. When the expression completes, the result will also be printed. So for the first expression

d = drop @ (2,3)

the program would print

Interactive cmd: d = drop @ (2,3)

Interactive cmd val (DROP): Drop[Pad(2,3), 0.5 μl of unknown]

This tells you that the result of the expression was a drop and that the value was the drop at pad (2,3), presumed to contain 0.5 μl of an unknown reagent.

If the expression contains an error, but it is reasonably clear what the mistake was, it will still be executed. So, for the input

d = drop @ (2,3

the program would print

Interactive cmd: d = drop @ (2,3

line 1:15 missing ')' at '<EOF>'

Interactive cmd val (DROP): Drop[Pad(2,3), 0.5 μl of unknown]

If the interpreter cannot figure out what you meant or if you try to combine objects that cannot be combined (e.g., asking to multiply a drop and an integer), it will say

Expression contained error, not evaluating.

In the case of a type error, the error message will tell you what sorts of things it could have done:

Interactive cmd: d\*3

line 1:0 Cannot compute DROP \* INT: d \* 3

expected one of:

FLOAT \* FLOAT -> FLOAT

FLOAT \* TICKS -> TICKS

FLOAT \* TIME -> TIME

FLOAT \* VOLUME -> VOLUME

INT \* INT -> INT

TICKS \* FLOAT -> TICKS

TIME \* FLOAT -> TIME

VOLUME \* FLOAT -> VOLUME

Expression contained error, not evaluating.

Interactive cmd val (NONE): None

Typing expressions

First off, yes, the input text box is terrible. There is no line wrap and no clipping at the box boundaries. The display package[[5]](#footnote-5) that is being tortured into doing unnatural things for this UI really isn’t designed for this sort of thing. At some point, it will be replaced. But for now, it’s good enough to type shortish expressions. For anything more complicated, you’re almost certainly going to want to use a text editor (e.g., Emacs, Notepad, Eclipse) to create a file of macro definitions that you can refer to. This file will be read in and executed when the program is run if you give it as the value following the --macro-file command-line argument.

To help a bit, the following are available:

* The left and right arrows allow you to move forward and backward within the text. Sometimes you have to click in the box first. I don’t know why.
* The up and down arrows scroll backward and forward through the expression history. This is useful when you typed something wrong and want to go back and fix it or when you want to go back and repeat an earlier command, perhaps with modifications.
* Ctrl-X and Ctrl-C cut and copy the whole text in the box. There is no way to select partial text. This can be useful both to pull pieces from earlier commands to paste into one you’re building and also to grab snippets to paste into a file using a text editor.
* Ctrl-V pastes the contents of the system clipboard at the current cursor position.

The macro language

The DMF macro language allows you to define complex actions that can be used in interactive expressions. It is not (yet, at least) a full language––for example, there are (ditto) no looping constructs other than those built into the drop motion operators and no user-defined types—and it is not expected that full protocols will be built with it,[[6]](#footnote-6) but it should make the process of interactively testing the board easier.

Overview

The language has been designed as a sort of “warm-up exercise” for a follow-on language for building protocols, and that language is aimed at non-programmers, so the macro language has a couple of features that are a bit strange for a programming language.

**Note: This document is being written before I leave on vacation, so I’m documenting the language as it has currently been released. This necessarily means that there are quite a number of places where I’m looking at it and saying, “That needs to be fixed.” Since time is short, I’m documenting things as they are, even if the fix looks to be trivial. I’ll update the document when I get back and fix things.**

Natural language-ish

First of all, there are constructs that are more common in English than in programming languages. For example, rather than saying something like w.exit\_pad to obtain the exit pad for well w (the pad on the board proper immediately in front of it), in this language, you say w’s exit pad. The ’s operator is used to obtain an attribute of an object, and some built-in attributes and operations are specified with multiple words.

Similarly, there are sometimes “noise words”—optional words that allow you to be more verbose. For example, the action of turning on a pad’s electrode can be specified as turn on or simply as on. Removing a drop from the board is minimally remove, but it can also be remove from board or remove from the board.

Along the same lines, there are often multiple ways to say things. The direction toward the top of the board can be up or north. If dir holds that direction as a value, you can get the value right (or east) by saying dir turned right or dir turned clockwise. The word column can be abbreviated as col. A time can be specified as 2 s or 2 sec or 2 seconds.

Finally, while you are required to specify the type of each parameter to a defined macro, giving them names is optional. If you prefer, you can simply refer to them as, e.g., the drop (or simply drop) or pad 3.

DMF-relevant types

In addition to normal programming types like integers, floating point numbers, and boolean values, the macro language also supports types for concepts that are important in the domain. Drops, pads, and wells are first-class objects, as are directions (e.g., right), deltas (e.g., 3 up), time (e.g., 100ms), and motions (e.g., to row 5)

Preliminaries

Before getting into the meat of the language, here are a couple of concepts that don’t really fit elsewhere.

Comments

Comments come in two forms. First, everything from // to the end of the line is ignored:

n = n+1; *// add one to n*

Second, everything between /\* and \*/ is ignored:

n */\* the new value \*/* = n */\* the old value \*/* + 1;

The second form of comment can span multiple lines:

*/\**

*\* Add one to n*

*\*/*

n = n + 1;

Note that the second form does not nest. It ends at the first \*/ it sees.

Variables

Variables start with a letter or an underscore and contain letters, numbers, and underscores, optionally ending with a question mark. If a variable name begins with an underscore, in must be followed by a letter, number, or underscore.

These are all valid variable names:

a

test

n\_2

\_private

is\_waste?

The only exception is that generally things that could have been variable names but are used as keywords are not recognized as names. Most of the time there’s no reason they couldn’t be (i.e., it wouldn’t make things ambiguous), and I plan on making most of them usable as names, but for now only some of the short keywords (e.g., s, ms, x, and y) are carved out as exceptions. If the parser complains, choose another name.

Scoping

At the moment, variable scoping is…not exactly *broken*, but perhaps a bit counterintuitive. It will be fixed. I’ll try to describe here how it works.

**Note: You probably don’t need to read this section unless you start getting error messages about undefined variables or variables that are defined when you don’t expect them to be and you don’t understand what’s going on.**

At base, the macro language uses static scoping. The top-level (global) scope contains names defined at the top level of imported files and in the UI’s text box. Within a macro definition, the parameters form a nested scope, and each nested block starts its own scope. This is all as one would normally expect.

The problem arises because of a (poor) design decision, namely that there is no actual syntax for declaring a variable other than the parameter to a macro. Rather, variables are declared when they are first assigned, *unless there’s a variable with that name already in scope*, in which case that variable is used.

This means that in

x = 5;

*// hundreds of lines skipped*

some\_func = macro(d: drop) {

x = d’s column;

// …

};

when some\_func is called, the global x will be clobbered, which probably isn’t what was intended. Worse (or, perhaps not, because it will at least give you an error), if the global x had been initially assigned something that couldn’t be assigned an integer, the interpreter would have complained.

Similarly, if you try to do something like

test = macro(drop) {

if the drop’s row > 2 {

d = down;

} else {

d = up;

}

drop : 1 in direction d;

};

you will get an error complaining that d was not defined in the last line. This happens because each of the two assignments determines that d is a new variable, created in the branch’s local scope. This can be solved by writing, e.g.,

test = macro(drop) {

d = up;

if the drop’s row > 2 {

d = down;

}

drop : 1 in direction d;

};

Now d is defined in the top-level block of the macro, and that value is what will be modified in the if statement’s “then” clause.

A similar situation arises if you try to define recursive (or mutually recursive) macros, e.g.,

fib = macro(n: int) {

if n == 0 {

0;

} else if n ==1 {

1;

} else {

fib(n-1) + fib(n-2);

}

};

The name fib isn’t defined until the assignment, so within the macro, it’s undefined. This can be kludged around by adding a definition like

fib = macro(n: int) { 0; };

before the real definition. This tells the interpreter that fib is defined (as a macro taking an integer and returning an integer) at the global scope, and the real definition’s assignment overwrites the dummy one. Note that in the real definition, the last line simply says, “Use the definition of fib in the global scope”, which will be the real one when the function is called.

Expressions

For the most part, expressions will be described along with the types they apply to. For example, arithmetic expressions on numbers and time values, as well as addition of deltas to pads. Here we will touch on some of the more general concepts.

Attribute values

As mentioned above, attributes are accessed by using the ’s operator. For example, a pad p’s row can be obtained by p’s row.

In some cases, the object may or may not contain a value specified by an attribute. For example, a pad may or may not contain a drop. If you ask for p’s drop and pad p doesn’t have a drop, an error will be signaled. To guard against that possibility, you can say, e.g.,

if p has a drop {

p’s drop : up 2;

}

The has a (or has an) expression returns true or false depending on whether the object has a value for the attribute in question.

Conditional expressions

To choose one value or another based on a third, the if…else expression can be used:

x if x > y else y

The middle expression is evaluated first, and based on its value, either the first or last expression is executed. Note that the other expression is not evaluated.

The test expression must evaluate to a boolean value, and the two value expressions must evaluate to compatible types. (For example, one could be an integer and the other a floating-point number.) If they are compatible, but not identical, the value of the expression will be the most specific type that is compatible with both values.

Function calls

Function calls, whether to a macro or a built-in function or other function-like object (e.g., a delta), comprise a name and a comma-separated list of parameters in parentheses:

fn(1, d, x+2)

There is no provision (yet) for optional or named parameters, so the number and types of the parameters must match that of the function. (Except that, e.g., you can pass in an integer to a function that expects a float or a drop to a function that expects a pad).

Note that the “function” part of the call must be a name, which is looked up in the current scope, so all functions must be assigned to variables before being called.

Injections

A function that takes a single argument may be called using a simpler “injection” syntax, with the argument being “injected” separated from the function by a colon:

d : to row (n + d’s row)

n : squared

Note that unlike the full function call syntax, the function doesn’t have to be bound to a name. What happens is that first, the argument is computed, then the function is computed, and finally, the function is called with the argument as its only parameter.

The value of an injection is the value of the function call, unless the function doesn’t return an argument (e.g., a macro whose block is a parallel block, an if statement whose blocks don’t evaluate to the same type, or a sequential block whose last form doesn’t have a value). In this case, the value of the injection is the value of the argument.

Since the colon operator is left associative, injections can be chained:

d : left 2 : up 5 : pause 2 seconds : right 5

w : dispense : right 3

In the last form, w is a well and dispense is a macro that dispenses a drop and returns it. The dispensed drop (the result of w : dispense) is then walked three pads to the right and returned as the value of the entire injection chain.

When the left-hand side of the colon is itself a callable form, the behavior is a bit different (but hopefully intuitive). In this case, the colon returns a value which is itself a callable function which, when called, passes in its arguments to the left-hand function and then the result (or pass-through argument) to the right-hand function. For example

walk\_path = left 2 : up 5

makes walk\_path into a value that can be used as the injection target for a drop:

d : walk\_path

The left-hand side can take more (or less) than one argument, in which case, the full function call syntax must be used on the result. For example, if rectangle is a macro that takes a drop and two deltas describing the first two sides of a rectangular path,

rectangle\_plus = rectangle : 2 down

is a path that walks the rectangle and then continues on downward for two pads. It would be called like

rectangle\_plus(d, 3 right, 2 up)

Types

In this section, I’ll go over the various types of data that the macro language knows about. For each type, I’ll detail

* the name of the type (e.g., in macro parameter declarations),
* how you create a value of the type,
* what sorts of expressions you can perform on it, and
* what attributes it has.

Numbers

Numbers come in two flavors: integers (**int**) and floating-point numbers (**float**). Integers can be used any place that expects floating-point numbers.

Integer constants consist of a digit followed by zero or more digits or underscores:

0

52

5\_000

A floating-point number is an integer followed by an exponent (e.g., e-5) or by a decimal point, an optional integer, and, optionally, an exponent:

3.0

0.5e5

1e7

2.

12.123\_456

Note that a leading digit is required. .5 (point five) is not a legal floating-point number.

Numbers can be added (+), subtracted (-), multiplied (\*) and divided (/). For addition, subtraction, and multiplication, if both arguments are integers, the result is an integer, otherwise it is a floating-point number. The result of division is always a floating-point number, even if the arguments are both integers.

To convert a floating-point number to an integer, use one of

round(n) *// rounds n to the nearest integer*

floor(n) *// returns the largest integer less than or equal to n*

ceil(i) *// returns the smallest integer greater than or equal to n*

Numbers may be compared using ==, !=, <, <=, >, and >=. These operators return boolean values.

Boolean values

Boolean (**bool**) values represent truth or falsity. Constant truth may be specified as True, true, TRUE, Yes, yes, or YES. Similarly, constant falsity may be specified as False, false, FALSE, No, no, or NO.

If b1 and b2 are boolean values, not b1 returns the opposite of b1’s value, b1 and b2 returns true if both b1 and b2 are true, false otherwise, and b1 or b2 returns false if both b1 and b2 are false, true otherwise. Note that and and or are “short circuiting” operators. If the interpreter can tell based on the value of b1 what the value of the entire expression will be, b2 is not evaluated.

Directions

Directions (**direction** or **dir**) are represented by the constants up, down, left, right, north, south, east, and west.

Given a direction d, you can find related directions by using

d turned right

d turned clockwise *// the same*

d turned left

d turned counterclockwise *// the same*

d turned around

Strings

Strings are delimited by double quotes:

“This is a string.”

They may not contain newlines or carriage returns. Escape sequences are prefixed by backslashes:

|  |  |
| --- | --- |
| \t | tab |
| \r | carriage return |
| \n | newline |
| \” | double quote |
| \\ | backslash |
| \u*HHHH* | Unicode character with four-digit hexadecimal code. Letters can be upper- or lowercase. |

Some examples:

“Multi\nline\nstring”

“String\twith\ttabs”

“lowercase mu: \”\u00B5\””

Strings are concatenated by adding (+) them together. A number can also be added to a string.

Deltas

Deltas (**delta**) represent a number of steps (an integer) in a particular direction. The number of steps for a delta d can be found as

d’s distance

and the direction can be found as

d’s dir

d’s direction

Deltas in a constant direction are formed by, e.g.,

5 up

right 3

The direction can come either first or last, but there is a subtle distinction[[7]](#footnote-7)—the form with the distance first binds tighter than arithmetic operators, while the form with the direction first binds weaker. So

right n + 2

means n+2 steps to the right, while

p + 2 up

means 2 steps up from pad p. (That is, it adds the delta 2 up to the pad value p.)

As an alternative, you can use rows as a substitute for up in the distance-first form:

p + 2 rows

and, similarly, columns (or cols) as a substitute for right. When the distance is the constant 1, the keywords row, column, or col may (but need not) be used.

When the direction is computed at runtime, the in direction (or in dir) operator is used, e.g.,

2 in direction w’s exit dir

Pads

Pads (**pad**) represent spaces in the electrode grid on the board proper (i.e., not in the wells). Pads are characterized by a column (x coordinate) and row (y coordinate), both integers. The lower left-hand pad in the grid is coordinate (0,0), and the numbers increase upward and to the right.

A pad can be specified by a comma-separated pair of integer expressions in parentheses:

(2, 5)

(n, x+2)

It’s row and column (as integers) can be obtained by

p's row

p’s y coord

p’s y coordinate

p’s column

p’s col

p’s x coord

p’s x coordinate

A pad may or may not currently have a drop on it. If it does, the drop can be obtained as

p's drop

To test whether there is a drop there, use p has a drop.

A pad may or may not be the exit pad for a well. If it is, the well can be obtained as

p's well

To test whether the pad is an exit pad, use p has a well.

A delta may be added to or subtracted from a pad to get another pad the specified number of steps in the specified direction, for example

p + 2 right

p + (2\*n) down

p - 2 rows

Note that there is no guarantee that the resulting pad is actually on the board or a pad that can be controlled if it is, and there is (as yet) no way to test.

Electrodes

Pads (along with well pads, described below) are a subtype of “electroeds” (**electrode**). The primary feature of a component is that it has an electrode, and you can ask for the electrode’s current state by[[8]](#footnote-8)

p's state

Its electrode state may be modified by injecting the pad into a special action function:

p : turn on *// or* p : on

p : turn off *// or* p : off

p : toggle state *// or* p : toggle

Drops

Drops (**drop**) represent physical drops on the pads of the board.[[9]](#footnote-9) A drop has a current pad, which can be obtained by

d's pad

or simply by using d, as drops can be provided anyplace that pads are expected. This means, that you can say, for example,

d + 1 right : turn on

to turn on the pad one pad to the right of the drop’s current pad.

Drops also have an associated volume and an associated reagent, but you can’t currently access or modify them.

Drops are created by

drop @ (2,3)

drop at p + 2 right

(Either @ or at may be used.) A drop’s identity is maintained even as it moves, so after

d = drop @ (2,3)

d : right 2

d’s pad would now be (4,3) and

(4,3)’s drop == d

would be true.

As that last example illustrates, a drop can be injected into a delta, and this will cause the drop to move the specified number of steps in the specified direction.

A drop can also be asked to move to a specified row or column:

d : to row 5

d : to col 3 *// or* to column 3

It can also be asked to move to a specified pad:

d : to p + 2 up

In this case it will first walk to the target row and then to the target column. That is, this is equivalent to

\_p = p + 2 up

d : to row \_p’s row : to column \_p’s column

If a drop is no longer on the board (e.g., because it has entered a well), it can be removed from the model by saying

d : remove from the board

d : remove from board

d : remove

It is an error to try to use a drop that has been removed.

Wells

Wells (**well**) are identified as

well #2

well #(n+1)

(There is (as yet) no way to get a well’s number from a well object.) The wells on the Joey board are numbered from 0 through 3 down the left-hand side and 4 through 7 down the right-hand side. On a Wombat board, only wells 2, 3, 6, and 7 are available.

Wells have the following attributes:

w's gate

w’s exit pad

w’s exit direction *// or* exit dir

The exit pad is the pad on the grid immediately in front of the well. The gate is the well pad immediately adjacent to the exit pad. And the exit direction is the direction from the gate to the exit pad.

Well pads

Well gates (**well pad**) represent the electrodes in the well. Like pads, they are components (see above) and have state which can be queried or modified. Unlike pads, they don’t have coordinates, but they do know their well:

wp's well

A well’s gate is obtained by

w's gate

Its interior pads are obtained by

w[n]

where n is an integer identifying the well pad. On the Joey board, well pads 0–2 are (from top to bottom) the well pads in the column adjacent to the gate. The next column contains well pads 3–5, and then the larger well pads 6, 7, and 8.

Note that the analogous interior pads on each side of the Joey board share their electrode state, so after saying well #2[0] : on, it will be the case that well #3[0] will also be on. Note further, however that these two well pads are not the same object, and they have different values for their well attributes.

Time, ticks, delays, and pauses

It is sometimes useful to be able to specify that execution pause for a certain amount of time, and that can be done by saying

pause 100ms;

pause 3 ticks;

As can be seen, the pause can be specified in elapsed time (**time**) or clock ticks (**ticks**). In the latter case, the pause is subject to the system clock being stopped, started, and stepped. Both time and ticks values can be added and subtracted, and both can be multiplied and divided by floating-point numbers. They can also be compared to like values.

Time literals are specified by a floating-point expression followed by a time unit, which can be one of

s, sec, secs, second, seconds

ms, millisecond, milliseconds

Ticks literals are specified by an integer expression followed by ticks or tick.

Both time and ticks are subtypes of a more general delay type (**delay**) type, which is what pause expressions actually take and which can be specified as the type of a macro parameter.

Note that pause expressions mean different things when used as statements and when used in expressions. As a statement,

pause 3 ticks

means that the current thread of control should pause for three clock ticks. When used inside of an expression, it creates a functional object which takes one argument (of any type), pauses, and returns its argument. So, if you write

d : right 2 : pause 3 ticks : left 2

drop d will move two steps to the right, pause for three clock ticks, and then move back to its original position.

Statements

The set of statements in the macro language is rather impoverished (it doesn’t for example, have any looping constructs), although it does have the notion of doing several things in parallel.

Expression statements

The most basic statement consists of an expression followed by a semicolon. This is primarily used in two situations.

First, an expression consisting of a function call or injection may be used for its side effects such as changing pad state or moving drops.

Second, an expression statement as the last statement in a sequential block becomes the value of that block.

Assignment statements

An assignment statement has the form

var = expression;

Note the semicolon at the end. This is required even when the value expression is, e.g., a macro definition, which ends with a closing brace.[[10]](#footnote-10)

As discussed above, an assignment will assign to the variable with that name, if there is such a variable visible in the current scope. If there is not, the first assignment seen is taken to be a declaration of such a variable in the current scope. This declaration also establishes the type of the variable[[11]](#footnote-11), which can’t be changed.

Note that assignments are (at the moment) statements and not expressions. There is (currently) no way to assign a value to a variable in the middle of an expression.

Sequential blocks

Sequential blocks are written as an opening brace {, a sequence of statements, and a closing brace }. The statements are executed one after the other, and the value of the block is the value of the last statement. For example,

{

p = d’s pad;

d : right 2;

p : toggle;

d’s y coordinate;

}

First d’s pad is stored as p. This variable will be local to the block (unless p has already been established as a variable outside the block). Next d is walked two pads to the right. Then p’s electrode is toggled. And finally, the y coordinate of d’s current pad is established as the value of the block.

Parallel blocks

Parallel blocks look like sequential blocks, except that instead of using braces, they use double square brackets [[ and ]].

In a parallel block, all of the statements are scheduled to happen at the same time. For those that involve modifications to the board, the first modifications in each statement will happen on the same clock tick. (Subsequent modifications do not necessarily line up.) The parallel block is considered completed when all of its statements are completed. A parallel block has no value.

Note that the contained statements race each other, so if two statements modify the same variable, unless care is taken, it will be impossible to predict which modification happens first. That is, in

[[

x = 5;

x = 6;

]]

after the block is done, x will be either five or six, but you can’t predict which, and it might not be the same each time.

Parallel blocks are especially useful when the statements are electrode state changes. For example, in

[[

w[0]: off; w[1]: off; w[2]: off;

w[3]: off; w[4]: off; w[5]: off;

w[6]: on; w[7]: on; w[8]: off;

w's gate: off;

]]

All of well w’s well pads are set to known values in a single clock tick. Similarly, in

[[

d1 : right 4 : up 2;

d2 : right 8 : down 2;

]]

two drops are walked, in parallel, along two separate paths. The block ends when the longer path is done.

Parallel blocks and sequential blocks are often combined, especially for things like dispensing sequences from wells:

{

[[ w[3]: on; w[4]: on; w[5]: on; ]]

[[ w[0]: on; w[1]: on; w[2]: on; ]]

w's gate: on;

w's exit pad: on;

[[ w's gate: off;

w[0]: off;

w[1]: off;

w[2]: off;

w[3]: off;

w[5]: off;

]]

[[ w[0]: on; w[1]: on; w[2]: on; ]]

w: ready;

drop@w's exit pad;

}

This putative dispensing sequence is a sequential block containing eight steps. The first six, including four parallel blocks, modify the state of the well’s pads, the seventh injects the well into a macro that gets it back to a “ready” state, and the final statement creates a drop at the well’s exit pad and returns this drop as the value of the sequential block.

Blocks can also be combined the other way, with a set of sequential blocks performed in parallel.

If statements

Conditional code is specified by means of if statements:

if x < d’s column {

d : to column x;

} else if y < d’s row {

d : to row y;

} else {

d : to p + 2 down;

}

There can be any number of else if clauses (including zero), and the final else clause is optional.

Note that each of the clauses must be a block (not a statement). It can be a sequential block or a parallel block, and the block may only contain a single statement (or even no statement), but it must be a block.

If there is an else clause and all clauses evaluate to objects of the same type (or which have a common supertype, e.g., integer and floating point or drop and pad), the value of the if statement will be the value of the executed clause. Otherwise the if statement has no value.

Macros

Macros are functional objects that can be applied to arguments. They are created by expressions consisting of

* the keyword macro,
* a header, which describes the types and, optionally, names of the parameters to the macro, and
* a body, which may be a (parallel or sequential) block or an expression.

Note that macros are values. Typically, they will need to be assigned to a variable in order to be useful. Note in particular that this assignment will require a final semicolon.

Examples

Some examples:

abs = macro(x: int)

x if x >= 0 else -x;

manhattan = macro(pad 1, pad 2) {

rdiff = abs(pad 1’s row - pad 2’s row);

cdiff = abs(pad 1’s col - pad 2’s col);

rdiff + cdiff;

};

ready = macro(w : well)

[[

w[0]: off; w[1]: off; w[2]: off;

w[3]: off; w[4]: off; w[5]: off;

w[6]: on; w[7]: on; w[8]: off;

w's gate: off;

]];

col\_first = macro(drop, pad) {

drop : to col (the pad’s col) : to row : (the pad’s row);

}

The first defines a macro abs, which computes the absolute value of an integer. This is so simple, it’s reasonable to use a single expression for its body, although it could also be done as a single-statement block. The single parameter is declared to be an integer and is named x.

The second computes the Manhattan distance between two pads, calling the abs macro. Its two parameters are both pads and they are declared to be referred to by number rather than by name.

The third macro encapsulates the parallel block example we saw above, which gets its well parameter (named w) into a ready state.

The final macro does a column-first walk to a target pad. Here, neither of the parameters are named. In the body they are simply referred to as “drop” and “the pad”. (Either form is acceptable.)

Closures

Variables referenced within macros that are not specified as parameters or first assigned within their bodies are taken to refer to variables of the same name in the surrounding context at the time the macro is defined. In particular, when macros are defined within macros,[[12]](#footnote-12) they can refer to local variables within the enclosing macro, and these bindings persist even after the enclosing macro has finished executing. So, you can do things like

adder = macro(n : int) {

macro(x : int) {

x + n;

};

};

add\_one = adder(1);

add\_five = adder(5);

Every time adder is called, it returns a new macro which adds its value to the value that was passed in to adder. So, add\_one(7) will return eight, and add\_five(7) will return twelve.

It’s even possible to maintain state this way:

counter = macro() {

n = 0;

macro() {

n = n+1;

n;

};

};

times\_called = counter();

When counter is called to define times\_called, creates a new variable n, which its value will read. The first time times\_called is called, it will increment this n and return one. The second time, it will return two, etc.

1. Note that it cannot be abbreviated to “disp”, as that will try to run the “dispense” task. [↑](#footnote-ref-1)
2. Plus arguments specific to the task. The display task has no additional arguments. [↑](#footnote-ref-2)
3. The code is written in Python, so the rate is somewhat approximate, but it should be stable enough for the purposes of moving drops around. [↑](#footnote-ref-3)
4. For Wombat boards, it appears that limitations of the OpenDrop controller make it impossible to meaningfully set the clock interval less than 100 ms. [↑](#footnote-ref-4)
5. Matplotlib. [↑](#footnote-ref-5)
6. There is a whole underlying Python API for that whose description is outside the scope of this document. [↑](#footnote-ref-6)
7. It represents my intuitions. It may be a bit too subtle. [↑](#footnote-ref-7)
8. Unfortunately, it appears that you can’t do anything with this at the moment, as there’s no way to either test against a constant value or use it to set the state of another pad. Sigh. [↑](#footnote-ref-8)
9. Or, more accurately, they represent *modeled* drops believed by the program to be on the board. [↑](#footnote-ref-9)
10. If you forget the semicolon, the interpreter will probably forgive you, but it will print a warning. Note that assignments in the UI’s text box do not need them. [↑](#footnote-ref-10)
11. The established type is the type of the expression, which may well be wrong (e.g., when the expression returns an integer, but the variable should really be thought of as floating-point). In the future, you will be able to specify the type of the variable being declared and also to declare variables that shadow those visible from external scopes. [↑](#footnote-ref-11)
12. Yes, this is possible. No, you will probably not need this. [↑](#footnote-ref-12)