A Sketch of a Multilevel API for Microfluidics Platforms

Evan Kirshenbaum

*Sunday, April 25, 2021*

Introduction

In this document, I aim to sketch out my current thinking on an API for digital microfluidics platform.[[1]](#footnote-1) This is different from my work on the Thylacine languages, although I suspect that something like it will likely sit underneath them. I call it a “sketch” because I mean to lay out the conceptual objects and operations and not work out the fussy details of the syntax in any given language. Indeed, I suspect that it will likely be presentable in pretty much any language, including interpreted languages like Python.

I’m describing this as a *multilevel* API, because it presents operations at several different levels, which will be of use to different applications and interactive users. Roughly, these layers are:

* Level 1: the **device level**. Operations at this level include turning on and off electrodes and magnets, setting desired levels for heaters, and starting dispensing sequences for wells.
* Level 2: the **drop level**. Operations at this level include dispensing a drop, moving it to a neighbor pad, merging it with another drop, and heating a well for a period of time.
* Level 3: the **path level**. Operations at this level involve walking a train of drops along a path from a well, interacting with drops on other paths, and terminating in other wells.
* Level 4: the **operation level**. Operations at this level talk in terms of quantities of reagents, mixing them with other reagents, buffering, heating them, and the like. This level assumes that the quantities of particular reagents in the wells have been specified. The library chooses the paths to be used at this level.
* Level 5: the **recipe level**. Operations at this level deal with planning and running recipes, which contain operation-level operations and a partial order of temporal constraints between them. At this level, operations may be reordered or run in parallel (subject to ordering constraints), and the planner may determine what should be the initial contents of each well.

Each of these levels (along with its operations) will be described in more detail below.

What this document is not

I want to be clear up front that this is an early design document. It is not a manual or tutorial. It isn’t even a specification. If we proceed with this, those will come later.

It also isn’t tied to any language. For illustrative purposes, I’ve used a Java-like syntax, but don’t read too much into that. It should be able to present in almost any language, and the details of the presentation will likely differ. My guess is that there will be a core of the library written in C++ because that’s the language that’s easiest to put underneath other languages.

Finally, it doesn’t yet touch on what goes underneath the library, the part that needs to be provided by (or specialized to) a particular board manufacturer. That’s my next task.

Types

Before getting to the operations, I want to describe the types that they operate in terms of.

Basic types

In addition to the normal programming types (e.g., integers, floating-point numbers, strings, Boolean values). The following will be available:

* ***Binary states*** are either **on** or **off**.
* ***Directions*** are **north** (or **up**), **south** (or **down**), **east** (or **right**), **west** (or **left**), **northeast**, **northwest**, **southeast**, and **southwest**.
* ***XY coordinates*** have an integers *x* and *y*[[2]](#footnote-2) and are used for indexing into the *board*’s pad array. An XY coordinate has neighbors in the four directions and can tell whether it is within the bounds of the board’s pad array.
* ***Regions*** are rectangles of XY coordinates specified by an XY coordinate *origin* at their (typically) lower left corner and a *number of rows* and *columns*. A region can report its *height*, *width*, *lower left* coordinate, and *upper right* coordinate, and it can tell whether an XY coordinate (or pad) is within it. A region is *empty* when its height and width are both zero.
* Dimensioned typessuch as ***volume***, ***time***, and ***temperature***, as well as, perhaps, rates (e.g., temperature/time for heating rate) are specified by multiplying floating point numbers with *units*.[[3]](#footnote-3) Like quantities can be compared and added, and quantities can be multiplied by numbers. Units will be provided for at least, e.g., **µl** (or **ul**), **ml**, **ms**, **s**, and **K**.
* ***Reagents*** have a *name* as well as possibly minimum and maximum storage temperatures. Reagents can optionally specify a *composition*, which is a mapping from *chemicals* to *concentrations* (by mass, molarity, or volume). A special **waste reagent** is available for unimportant products that can be combined together in waste wells or flushed. The **unknown reagent** is used at the device, drop, and path levels when the initial contents of wells are unspecified and at all levels when the product of mixing two reagents is unspecified.
* ***Beads*** have an optional *name* of the substance bound to them. It’s possible that they may need size and/or mass attributes, but I haven’t seen a need for it yet.
* ***Liquids*** are a *volume* of a *reagent*[[4]](#footnote-4) and may contain *beads*. Note that this is a particular quantity of the reagent, and the volume (and possibly reagent) of a liquid can change over time, as can the presence or absence of beads. It may be useful to be able to specify the concentration of the beads. I’m assuming at the moment that a given liquid contains at most one type of bead.

The volume may be flagged as being *inexact*, in which case it will be taken as a lower bound of the actual content, or as *unknown*. This will be useful for the specification of liquids in wells filled by hand or in wells or pipettor reservoirs asserted to contain an unknown (but sufficient) quantity.

* ***Drops*** (or perhaps ***droplets***) are (or have) *liquids* (expected to be of small volume) that are associated with a specific *pad* (see below). That is, a drop knows where it is, and where it is can change over time.
* ***Futures*** are used to obtain values for asynchronous requests (including requests gated by the board’s clock), to determine whether such requests have completed, to wait for them to complete, and, ideally, to obtain an estimate of the time remaining and to register a callback to be called upon completion. Each superstrate language will have its own notion of dealing with asynchrony and synchronization,[[5]](#footnote-5) and the intent is to blend in with the languages model.

Platform component types

A given platform consists of a ***board***, which contains an array of ***pads*** and a collection of ***wells***. It may also contain things like ***heaters***, ***chillers***, ***magnets***, ***cameras***, ***detectors***, ***thermometers***, and ***pipettors***.

In most cases, the creation of objects of these types will be the responsibility of the manufacturer of the board by means of a straightforward configuration API. Both users and the recipe-level planner will need to make use of the configured constant values.

The board

The board is characterized by

* A rectangular array of *pads*, indexed into by XY coordinates, where x is the column and y is the row, with x and y running between a *minimum column* and a *maximum column*, and y running between a *minimum row* and a *maximum row*. (I expect that the minimum row and column will likely be zero, but I want to allow 1-based indexing to match a manufacturer’s documentation.)
* A *minimum* and *maximum* drop size. Drops between these (inclusive) bounds can be counted on to move properly as a cohesive unit and to stay within the bounds of a pad.
* A *basic drop size*, the amount dispensed by (default) wells. The board will provide a **drops** volume unit of this size.
* The *pad motion time*. This is the minimum amount of time after a neighboring pad’s electrode is turned that you can be confident that a drop has moved to the neighbor and can be induced to move further.[[6]](#footnote-6) This can also be thought of as the board’s *clock rate*, and the board will provide a **ticks** time unit of this size.
* Lists of *wells* and other resources such as *magnets*, *heaters*, *chillers*, *cameras*, *detectors*,and *pipettors*. Each of these will be accessible by indexing into the appropriate list and specific boards may provide other means of accessing (e.g., a data member for a unique camera or heater, or a map from names to wells).
* A collection of *dead regions* containing pads which must be avoided in routing.

Pads

A pad is characterized by

* The XY coordinate of its index into the pad array.
* Its possible association with a *well* (as the well’s exit pad), a *magnet*, a *heater*, and a *chiller*. Note that aside from the well, these other resources may be shared with other pads or, possibly, wells.
* Whether or not it is *nonexistent*. A nonexistent pad has no electrode and cannot be occupied by a drop. It usually results from the presence of a physical support where the pad would otherwise be.
* Its *neighbors* in any of the eight *directions*. A neighbor will be a pad if the resulting coordinate is on the board and refers to a pad that is not nonexistent. Otherwise, it will be a null value.

The transient state of a pad includes

* The *binary state* of the pad’s electrode.
* The *drop* that occupies the pad, or null.
* The *dried liquid* that occupies the pad, or null.
* Whether the pad is believed to be *broken* and should be avoided.

Wells

A well is characterized by

* Its *capacity* (a volume).
* Its *exit pad*. The location where drops from this well will be dispensed.[[7]](#footnote-7)
* Its *dispensed volume*, the size of the drop dispensed by the well (usually expected to be the board’s basic drop size).
* Its *dispensing time*, the amount of time between requesting that the well dispense a drop and the next time a dispensing request can be made. Typically, this will be an integer multiple of the board’s clock tick size. I am assuming that this is also the amount of time between requesting a drop be dispensed and the resulting drop being on the well’s exit pad and ready to move. (If this is not the case, we may want to be able to specify both.)
* Its *absorption time*, the amount of time between requesting that the well absorb a drop from its exit pad and the next time an absorption request can be made.
* Its possible association with a *heater*, a *chiller*, and a *pipettor*. Note that these other resources may be shared with other wells (and possibly pads). For a pipettor, there may be information required to identify the well to the pipettor (e.g., which pipette to use).
* Whether the well is *voidable*. If so, when the well is believed to have dispensed all of its contents, it will be considered to be available to receive drops of a different reagent.
* A list of *well sections*, each of which is characterized by its well, its index in the list, and a set of wells that share the well section’s electrode state at that index, and a set of indices that share the well section’s electrode state in each well.[[8]](#footnote-8) Its current state includes the *binary state* of its electrode and, possibly (depending on whether the system models motion within the well), the drop that covers the electrode in the well.
* A list of *dispensing sequences*.[[9]](#footnote-9) These are functional objects that take a requested volume and perform device-level actions that result in a drop (ideally of the requested size) appearing on the well’s exit pad. The first element of every such list will be the well’s *default dispensing sequence*, which dispenses a drop of the well’s dispensing volume.

The transient state of a well includes

* Its *contents*, a liquid (and thereby a reagent and contained volume). Based on this, the *remaining capacity* of the well and the *drop availability* (the number of drops (not a volume) that the well can dispense, including a possible fractional remainder), can be computed. A well is *empty* when its drop availability is less than one.
* Whether the well is *available*. A well is available if it has never contained a liquid or if it is voidable and the volume of the liquid contained is exactly zero.

Magnets

A magnet is characterized by

* The *set of pads* affected by the magnet. Each such pad also carries an association with the magnet, so the magnet can be accessed either via the board’s magnet list (when its index is known) or via the pad’s magnet association.

The transient state of a magnet includes

* The *binary state* of the magnet.

Heaters

A heater is characterized by

* Its *maximum temperature*.
* The *heating rate*, in units of temperature/time. (If it turns out that the time it takes to get to a desired temperature is largely independent of the desired temperature, it may be better to go with a *heating time* instead.)
* The *recovery rate*[[10]](#footnote-10) (or *recovery time*). This characterizes the time taken for a heater to return to ambient temperature after it has been turned off.[[11]](#footnote-11)
* The *set of regions* of pads and *wells* affected by the heater. Each such pad and well also carries an association with the heater, so the heater can be accessed either via the board’s heater list (when its index is known) or via the pad’s (or well’s) heater association.
* An indication of whether the current temperature of the heater can be read, possibly accompanied by the *accuracy* of such a reading.

The transient state of a heater includes

* The *binary state* of the heater.
* The *target temperature* of the heater when it is on.
* The *current temperature* of the heater, when this is available.
* The *delay time* required for the heater to get to its target temperature or ambient temperature.

Chillers

Much like heaters, a chiller is characterized by

* Its *minimum temperature*.
* The *chilling rate*, in units of temperature/time. (If it turns out that the time it takes to get to a desired temperature is largely independent of the desired temperature, it may be better to go with a *chilling time* instead.)
* The *recovery rate* (or *recovery time*). This characterizes the time taken for a chiller to return to ambient temperature after it has been turned off.
* The *set of wells[[12]](#footnote-12)* affected by the chiller. Each such well also carries an association with the chiller, so the chiller can be accessed either via the board’s chiller list (when its index is known) or via the well’s chiller association.
* An indication of whether the current temperature of the chiller can be read, possibly accompanied by the *accuracy* of such a reading.

The transient state of a chiller includes

* The *binary state* of the chiller.
* The *target temperature* of the chiller when it is on.
* The *current temperature* of the chiller, when this is available.
* The *delay time* required for the chiller to get to its target temperature or ambient temperature.

Thermometers

A thermometer is characterized by

* The *maximum* and *minimum temperature* it can read.
* The *accuracy* of its reading.
* The *settling time* that it takes to make a reading.
* The *pad* or *well* that the thermometer reads the temperature of or that the thermometer measures ambient temperature.

The transient state of a thermometer includes

* Its *current reading*.

Cameras

A camera is characterized by

* Its *visible region* as well as a set of *occluded regions* that represent parts of the visible region that it can’t see due to obstructions.
* The set of *wells* that it can see.

The transient state of a camera includes

* The current image.

Detectors

“Detectors” refers to an amorphous class of capabilities for obtaining information about the current state of the board. Such capabilities will often be purely software (provided by the manufacturer or configured in by the user), but they may also be a view onto idiosyncratic hardware for a particular platform. Purely software detectors will use other, hardware-provided information, such as that provided by cameras.

With detectors, the user may be able to do things like determine the color of the fluid in a drop or well, the precise volume of fluid in a well, or the number of cells of a particular kind in a drop. The actual interface to a particular type of detector will be specific to that detector type.

Users will obtain detectors in one of three ways:

* by walking the board’s list of detectors and asking (e.g., via Java’s **instanceof** operator) whether the detector is of the desired class,
* by using a state variable provided by a particular manufacturer’s board interface, or
* by being the one to create the detector and install it in the board.

Pipettors

Pipettors are any means of adding liquid to wells or removing liquid from them. This includes by using a user interface to request that a human do so.[[13]](#footnote-13)

A pipettor is characterized by

* Its *fill rate* and *extraction rate*, expressed in terms of volume/time, assuming the pipettor is in position. (If the fact that the volumes are so small means that they don’t really matter, we could just use *fill time* and *extraction time*.)
* The number of *reservoirs* it contains along with their *capacities*.
* The presence or absence of a *waste reservoir* (or *drain*).
* A collection of sets of *simultaneous well groups*. This is useful for the case in which a single pipettor can simultaneously fill, e.g., the four wells on one side of a board or, by moving, the four wells on the other side of the board.
* The *movement time* required to switch between one simultaneous well group to another, assuming that no fluid transfer is in progress.

The transient state of a pipettor includes

* The *current well group* that it is positioned to use (if any).
* The *current contents* (as a liquid, i.e., a reagent and volume) in each of its reservoirs.
* Whether it is *currently transferring* fluid and an estimate of the *time remaining* until current transfers are complete.

Requests on a pipettor that require a different simultaneous well group than the current one will be queued and may be preempted by subsequent requests that can be satisfied by the current well group or one earlier in the queue.

Dealing with Asynchrony

Most operations that involve drop motion or that otherwise may take a nontrivial amount of time to complete (e.g., heating a well) come in up to five varieties:

* A *normal* operation pauses computation until the operation is complete, returning its value.
* A *requested* operation initiates the action and returns immediately, even though the operation isn’t complete. If the value is known at call time, it will be returned, otherwise the operation has no value.
* An *asynchronous* operation starts its operation asynchronously and returns a future that can be used to determine whether the operation is still in progress, to wait for it to complete, and to obtain its value after completion.
* A *gated* operation schedules its operation (or, at least, the first stage of it) to take place at the next *clock tick*. It returns a future that will complete just before the clock tick following the completion of the operation.
* A *gated request* similarly schedules the initiation of the operation to take place at the next clock tick. Its return value is the same as for the requested operation.

For example, consider the move() operation on the class Drop:

Drop move(Direction dir);

This will be accompanied by four other variants:

Future<Drop> async\_move(Direction dir);

Drop request\_move(Direction dir);

Future<Drop> async\_move(Direction dir);

Drop gated\_request\_move(Direction dir);

Future<Drop> gated\_move(Direction dir);

* Calling move(up) initiates the move by turning on the electrode of the upward neighbor of the pad the drop is currently located at, waiting the board’s pad motion time, turning off the board’s pad, and returning. This form will probably be most useful when typing commands interactively.
* Calling request\_move(up) initiates the move (by setting the electrode) and returns immediately. This will be useful when the programmer wants to control the timing of actions explicitly.
* Calling async\_move(up) similarly initiates the move but will return a future that will complete after the pad motion time has elapsed. The programmer is free to do other things before the drop has finished moving but will be able to tell when it is done.
* Calling gated\_request\_move(up) schedules the initiation to take place at the next clock tick and returns immediately. This will be most useful when the program wishes to make a number of changes to electrode states and have them all take place at the same time. (This is the OpenDrop model.)
* Calling gated\_move(up) similarly schedules the initiation to take place at the next clock tick, and the operation returns similarly returns immediately, but with a future that will complete just before the clock tick following the one in which the initiation takes place. This is most useful when using the clock step model when an operation will take multiple steps (e.g., dispensing a drop) and the resulting value is important.

The clock

The clock is used to control when gated operations have effect. Many boards[[14]](#footnote-14) have an underlying operation that sets all of their electrodes at once and so it makes sense to declare a number of changes and then make one call to the hardware. Similarly, it typically takes a known amount of time for drops to move (the board’s pad motion time) so it makes sense to defer these updates until that has happened.

The library has a clock, under control of the program, that makes that happen. Its current state includes

* Its *current interval time*, initially, the board’s pad motion time.
* Its *current tick number*, initially zero.
* A flag indicating whether it is *running*. This is initially false.
* Lists of callbacks to take place *before* and *after* each clock tick.

When the clock is not running, it can be advanced to the next tick by calling

void advance\_clock([Time min\_delay])

If the optional delay parameter is provided, it ensures that at least that much time has elapsed since the last tick.

The clock is started and paused by calling

void start\_clock([Time interval])

void pause\_clock()

If the optional interval parameter is given, it is used to reset the current interval time.

While the clock is running, ticks will occur (roughly) regularly according to the interval time. Any gated operations that occur between ticks will happen as of the next tick.

For each clock tick, three things happen. First, the list of callbacks to take place before the tick are run. Then the actual electrode changes are communicated to the hardware. Finally, the list of callbacks to take place after the tick are run.[[15]](#footnote-15)

Callbacks are registered by calling

void before\_clock\_tick([int n], Callback callback)

void after\_clock\_tick([int n], Callback callback)

If the clock tick number is omitted, these are relative to the next clock tick.

Finally, there are two methods that allow the program to pause until the next clock tick or a particular clock tick number.

void await\_clock\_tick([int n])

and until there are no operations in progress

void await\_idle()

If you want an activity of your own to count as an operation for purposes of determining idleness, bracket it with

void begin\_operation()

void end\_operation()

It will probably also be worthwhile to have calls to stop the clock when the system is idle and to run it until the system is idle.

Multi-tick operations

Some operations, for example, dispensing a drop or walking a train of drops down a path, may require a sequence of drop motions. When these are performed as gated operations, they initiate their first move and then add a callback for after the next clock tick to do the next, repeating until they are done.

It’s not clear how they should work when done non-gated. I see three possibilities:

1. They work the same as gated operations. This has the downside that if the clock is not running, the user will have to manually advance it for each step.
2. They work completely independently of the clock and add their own delays based on the current clock interval time.
3. They work as gated operations when the clock is running but as free running whenever the clock is stopped.

The last, while more complicated, is probably least likely to violate the Principle of Least Astonishment.

Level 1: The Device Level

At the device level, operations manipulate the components of the physical device, without regard to the presence or absence of liquids.

All of the information regarding device component types above, both the characteristic information and the current state are accessible either as data members of the component objects or as method calls (based on language norms and whether communication with the device is required).

Pads

The only thing you can do with a pad is to set its electrode’s state:

BinaryState set\_state(BinaryState new\_state)

BinaryState turn\_on()

BinaryState turn\_off()

BinaryState toggle()

These operations return the previous state, because why not?

The gated variants take place at the next time tick, and the normal, request, and asynchronous variants return immediately, after communicating with the board.

Operating on a broken pad throws an exception. Either a separate call or an optional parameter can be specified to suppress this for the rare cases in which, e.g., you want to use a broken pad’s electrode to try to back out a misaligned drop.

Operations on, e.g., the magnet at a pad are performed by asking the pad for its associated magnet and performing the operation directly on the magnet.[[16]](#footnote-16)

Wells

The primary things you can do with a well are to run the sequences to dispense and absorb a drop:

Drop dispense\_drop()

void absorb\_drop()

These are also drop-level operations, and so dispensing a drop returns the new drop object. When working at the device level, it can be ignored.

Dispensing a drop is considered to be complete when the drop is on the exit pad. Absorbing a drop is complete when another drop could be moved to the exit pad.

If the well is believed to be empty when dispensing a drop, the dispensing sequence will be performed anyway, and a drop object will be created on the well’s exit pad. Similarly, if a drop is to be absorbed but the well is believed to be full, the operation will proceed. This behavior was chosen primarily to support situations in which the initial quantities in wells are unknown or unspecified, and when the user may be adding or removing liquid without telling the system. The alternative forms

Drop dispense\_drop\_or\_throw() throws WellEmptyException

void absorb\_drop\_or\_throw() throws WellFullException, NoDropException

throw exceptions if the operation is expected to fail based on the current well state. Absorbing a drop also throws an exception if there is no drop currently at the well’s exit pad.[[17]](#footnote-17) These exceptions are thrown immediately, even for the requested, asynchronous, and gated variants.

To run an alternative dispensing sequence, call

Drop run\_dispensing\_sequence(int i, Volume desired\_volume)

Drop run\_dispensing\_sequence\_or\_throw(int i, Volume desired\_volume)   
 throws WellEmptyException

The volume provided here is a hint; the actual drop may be of a different size. For the checked variant, the exception will be thrown based on the volume that would actually be dispensed.

Magnets

As with pads, the only thing you can do with a magnet is to set its state:

BinaryState set\_state(BinaryState new\_state)

BinaryState turn\_on()

BinaryState turn\_off()

BinaryState toggle()

Heaters

With a heater, you can request that it heat to a certain temperature or turn off and return to ambient temperature:

void heat\_to(Temperature target) throws OutOfRangeException

void turn\_off()

Both of these operations complete when the desired temperature is believed to have been reached.

Chillers

Chillers are just like heaters but have a chill\_to() operation in place of heat\_to().

Pipettors

Pipettors can be asked to add or remove liquid from a well:

Volume add(Volume v, Well well, int from\_reservoir)

Volume remove(Volume v, Well well, int to\_reservoir)

Volume fill\_to(Volume target\_volume, Well well, int from\_reservoir)

Volume extract\_to(Volume target\_volume, Well well, int to\_reservoir)

Volume drain(Volume v, Well well)

Volume drain(Well well)

All of these operations are considered complete when the fluid motion is finished. The return value is the volume of fluid added or removed (or, at least, an estimate thereof).[[18]](#footnote-18)

The fill\_to() and extract\_to() forms specify target volumes and attempt to add or remove liquid to meet the desired level. The drain() forms specify removal of unimportant liquids. The pipettor gets to decide where to put it and is licensed to combine different drained liquids.

It is not an error to request extracting more liquid than is in the well. The returned volume should be the actual amount transferred. Similarly, if it is known that a request to add volume will exceed the capacity of the well, less may be transferred. If the quantity of liquid in the well is not known exactly, and it turns out the request won’t fit, I suspect that that’s the user’s problem (unless the pipettor can know when to stop).

The “requested” variants of these operations return values based on the current content of the well.

Level 2: The Drop Level

At the drop level, operations are concerned with the movement of individual drops, as well as for the management of liquids within wells, focusing on the reagents involved.

Liquid management

To associate a pipettor reservoir with a given reagent and specify the (possibly unknown) volume of that reagent currently there, pipettors support

void contains(int reservoir, Liquid liquid) throws NoSuchReservoir

Specifying the waste reagent licenses this reservoir to be used for drain() operations.

Similarly, wells support

void contains(Liquid liquid)

If the well is known to be empty but will contain that liquid, simply use a volume of 0 ml.

To identify wells containing a particular reagent, the board supports

List<Well> wells\_containing(Reagent, [Beads])

Well well\_containing(Reagent, [Beads])

The first form returns a (possibly empty) list of all such wells, while the second returns the first such well or a null value.

To maintain the level using a pipettor, wells support operations that mirror those supported by pipettors:

Volume add(Volume v)

Volume remove(Volume v)

Volume fill\_to(Volume target\_volume)

Volume extract\_to(Volume target\_volume)

Volume drain(Volume v)

Volume drain()

The reagent involved must be known, and the system searches through the board’s pipettors until it finds one that (1) has a reservoir associated with that reagent and (2) has the well in one of its simultaneous well groups.

If the reagent hasn’t been asserted by the time the well is asked to dispense a drop, it is held to contain an unknown quantity of the unknown reagent. If the reagent hasn’t been asserted by the time it is asked to absorb a drop (or if it is available), it is held to have contained nothing of the reagent contained by the drop.

If some operation performed on the well causes the reagent to change, the new one can be asserted by

void change\_reagent(Reagent new\_reagent)

If the well has the capability of heating its entire content, it supports

void heat\_to(Temperature temp, Time duration) throws NoHeater, OutOfRange

This operation completes when the temperature has been reached and then held for the duration.

To hold the temperature of the well at a particular temperature, the well supports

void heat\_to(Temperature temp) throws NoHeater, OutOfRange

void chill\_to(Temperature temp) throws NoHeater, OutOfRange

void return\_to\_ambient\_temperature().

Dried liquid management

To note that a pad contains a dried liquid, pad’s support

void contains\_dried(Liquid liquid)

The volume of the liquid represents the volume of the drop before it was dried, and any component concentrations in its reagent are taken relative to that volume.

To identify pads containing a particular dried reagent, the board supports

List<Pad> pads\_containing(Reagent, [Beads])

Pad pad\_containing(Reagent, [Beads])

The first form returns a (possibly empty) list of all such pads, while the second returns the first such pad or a null value. Once the dried liquid on a pad has been reconstituted,[[19]](#footnote-19) it no longer shows up in this list.

Drop motion

Drops are (typically) created by invoking a well’s dispense() (or run\_dispensing\_­sequence()) operation, as described above with respect to level 1. The resulting drop is created and associated with the well’s exit pad.

For the other end of its life, the well supports

void enter\_well() throws NoWell

void enter\_correct\_well() throws NoWell, WrongReagent

Both of these throw an exception if the drop is not currently occupying some well’s exit pad, and the second form also throws an exception if the drop’s liquid is incompatible with the well’s current content.

These operations are considered to have completed when another drop could move to the well’s exit pad.

After entering a well, the drop object has no volume and is not associated with any pad.

To move n pads in a particular direction, drops support

Drop move(Direction dir, [int n = 1], [bool allow\_unsafe\_motion = false])

throws NoSuchPad, UnsafeMotion

This operation throws an exception (immediately) if some pad along the proposed path does not exist (either because it is off the board or because it is a nonexistent pad) or if the pad is broken. The operation is considered complete when the drop is stably on the final pad.

The operation throws an UnsafeMotion exception if the motion would take it to a pad that is a neighbor of another drop. (Typically, what should be done here is to use one of the mix or reconstitute methods.) If this motion is correct, specifying the allow\_unsafe\_motion parameter suppresses the check.[[20]](#footnote-20)

As with most drop motion operations, this operations returns the drop itself, so operations can be chained. Note that the pad associated with the drop will not be changed until the drop actually moves, so care must be taken when using the drop object in conjunction with asynchronous, gated, or requested forms.

When trying to mix two reagents (or get beads to disperse within a drop), it can be useful to move the drop back and forth to create a vortex. I’m not sure of the best name for such movement, but I’m currently calling in “shuttling”, so drops support

Drop shuttle(Direction dir, [int n = 1]) throws NoSuchPad

This moves the drop one step in the given direction, then back, and repeats this n times. It raises an exception if the requested motion is impossible. The operation is considered complete when the n repetitions have finished.

To mix two drops, a drop supports

Drop mix\_in(Drop other\_drop,   
 [Direction shuttle\_dir, [int n\_shuttles = 1]],   
 [Reagent new\_reagent]) throws NoSuchPad

This operation merges the other drop into this one and, optionally, does a shuttle operation to perform the mixing. If the two drops are adjacent, this moves the other drop into this one.[[21]](#footnote-21) If they are one pad apart, it moves both of them into the pad between them and does the shuttling relative to that pad. Otherwise, it throws an exception.

If no exception is thrown, this drop’s volume is increased by that of the other drop and the other drop’s volume is reduced to zero and it loses its association with its pad.

If the new reagent is specified, the drop’s reagent is change to it. Otherwise, if the two drops do not have the same reagent, the drop’s reagent is changed to the unknown reagent.

The mixing operation is considered complete when the shuttles are done.

To simplify the case in which mixing is followed immediately by splitting, a drop supports

Drop mix\_and\_split(Drop other\_drop, [int n\_shuttles = 1],   
 [Reagent new\_reagent]) throws NoSuchPad

This mixes with a nearby drop as with mix\_in, but then immediately splits the resulting drop in two, moving each half back to the original drops’ positions. If a number of shuttles is specified, the shuttling takes place by repeating the merging and separating. After the operation, both original drops remain, with the new reagent (if specified, otherwise the unknown reagent) and the average of their volumes.

To split a drop into n parts, a drop supports

Tuple<Drop, Drop> split(Direction dir1,Direction dir2) throws NoSuchPad

Tuple<Drop, Drop, Drop>

split(Direction dir1,Direction dir2,Direction dir3) throws NoSuchPad

Tuple<Drop, Drop, Drop, Drop>

split(Direction dir1,Direction dir2,Direction dir3, Direction dir4) throws NoSuchPad

This operation splits the current drop into up to four equal parts in the given four directions, creating up to three new drops corresponding to each direction. In the resulting tuple, the first element is the resulting, smaller, drop. The operation is considered complete when all of the motions is complete.

To deal with magnets, drops support

Drop drop\_beads() throws NoMagnet

Drop pick\_up\_beads([Direction shuttle\_dir, [int n\_shuttles]) throws NoMagnet

Drop wash([Direction shuttle\_dir, [int n\_shuttles]) throws NoMagnet, NoSuchPad

The drop\_beads() operation turns on the magnet at the drop’s pad, throwing an exception if the pad has no magnet. The pick\_up\_beads() operation turns off the magnet at the drop’s pad, throwing an exception if the pad has no magnet, then does an optional shuttle to ensure that the beads are suspended. The wash() operation picks up the beads, does an optional shuttle, and then drops the beads.

The drop and pick up operations are considered completed at the next time tick for gated variants and after the clock interval otherwise, while the wash operation is considered completed after its pick-up phase is completed.

To reconstitute dried reagents, drops support

Drop reconstitute(Direction dir, Time sit\_for,  
 [Direction shuttle\_dir, [int n\_shuttles = 1]],   
 [Reagent new\_reagent]) throws NoSuchPad, NoDriedReagent

This operation moves the drop to the specified neighboring pad, which must contain a dried reagent, or an exception is thrown. It then sits for the specified time to allow the dried reagent to dissolve and optionally performs shuttles to mix it. The drop’s reagent is changed to the new reagent or the unknown reagent. The drop’s volume is not changed.

The operation is considered complete after the shuttles are finished or, if none are required, after the sitting is done.

Level 3: The Path Level

At the path level, operations are concerned with trains of drops being dispensed from wills, marching along paths, possibly mixing with drops in other trains, and eventually winding up in other wells.

To describe a train of drops, wells support:

Train train(Volume v, [int cadence = dispensing\_time])

This describes a train of drops dispensed from the well, where the aggregate is the requested volume. The optional cadence is the amount of time between each drop, which may not be less than the dispensing time. Wells also support

Train train()

which describes a train whose volume and cadence can be determined from its interaction with other trains.

The train class supports the following methods to describe what happens next:

Train walk(Direction dir)

Train shuttle(Direction dir, [int n])

Train drop\_beads()

Train pick\_up\_beads()

Train wash([Direction dir, [int n]])

Train mix\_in(Train other\_train, [Direction dir, [int n]], [Reagent result])

Train split(Direction dir1, Direction dir2, Path path2)

Train split(Direction dir1, Direction dir2, Path path2, Direction dir3, Path path3)

Train split(Direction dir1, Direction dir2, Path path2,

Direction dir3, Path path3, Direction dir4, Path path4)

Train reconstitute\_and\_dilute(Direction dir, Time sit\_for, double tolerance,

[Reagent result])

Train enter\_well()

Train extend(Path path)

To actually run the train, the board supports

void run(Train train1, …) throws …

This analyzes the trains, figures out the required cadence and delays when they are unspecified, and makes sure that the trains make sense and throws exceptions if, e.g., a train would require magnet operations at places that don’t have magnets or for drops that don’t have beads, mixing operations at places the paths don’t have drops that are sufficiently close, or entering wells at places that aren’t exit pads for wells, and otherwise runs the described actions. The run operation is considered complete when all of the drops have entered wells.[[22]](#footnote-22)

The reconstitute\_and\_dilute operation advances the train one step so that the lead drop is on a pad containing a dried reagent (or an exception is thrown) and sits for the specified time while the dried reagent dissolves. The operation then performs pairwise mixing and splitting between the drops (along the path) until the difference in concentration of the new reagent between the drops with the lowest and highest concentration, taken as a ratio, is no more than the specified tolerance.[[23]](#footnote-23)

Note that it is fine for trains to cross one another’s path as long as the drops aren’t close enough to interfere with one another.

For the special case of running a single train, trains support

void run() throws …

Level 4: The Operation Level

At the operation level, operations are performed on liquids themselves, with no specification of the pads or wells, and the library picks the paths to use subject to the assumption that all non-nonexistent pads are available for it to use.[[24]](#footnote-24)

The relevant methods supported by liquids are:

Liquid mix\_with(List<Liquid> others, [int n\_shuttles], Reagent result)

Liquid mix\_with(Liquid other, [int n\_shuttles], Reagent result)

Liquid heat\_for(Time duration, Temperature at\_temp)

Liquid cool\_to\_room\_temperature()

Liquid buffer(Reagent release, [int n\_shuttles],

List<WashStep> washes, [Reagent collector)

Liquid buffer\_and\_discard(Reagent release, [int n\_shuttles],

List<WashStep> washes, [Reagent collector)

Liquid reconstitute\_and\_dilute(Liquid dried\_liquid, Time sit\_for,

double tolerance, [Reagent result])

The mix\_with() operations mix the given liquid with one or more other liquids to form a final reagent, optionally performing shuttles at each mixture. The specified liquids must be known to be in wells, and the resulting liquid is also in a well. The resulting liquid is returned, so this can be chained to perform more level-4 operations. If there is more than one liquid to be mixed with, the library is licensed to perform the mixtures in multiple stage and in any order, and liquids in the list may be mixed together before mixing with the original liquid. The operation is considered complete when the entire resulting liquid is in its well.

The heat\_for() operation heats the liquid, either in a well or by marching droplets through a heated region. It returns the liquid for further chaining. The operation is considered complete when the heating duration expires.

The cool\_to\_room\_temperature() operation lets the liquid sit until it is measured or believed to be back to room temperature. It is considered complete when this determination is made.

The buffer() operations are the most complicated. They can only be performed on a liquid that contains beads. They go through the following steps for each drop:

1. Drop the beads on the magnet.
2. Perform any wash steps.
3. Have the release reagent pick up the beads and optionally shuttle to mix them.
4. Collect the beads using the collector reagent (if specified). If none is specified, use a wash reagent specified as discardable or the original liquid (for the \_and\_discard() form).
5. Move all liquids to wells.
6. Return the liquid containing the release reagent and beads.

Wash steps contain a reagent, an optional number of shuttles to perform, and an indication of whether the reagent can be discarded when done.

The reconstitute\_and\_dilute() operation uses the given liquid to reconstitute a dried liquid (which must be on the board), sit for the required time to let it dissolve, and performs pairwise mixing to allow the concentrations of dissolved liquid in all of the drops to be within a given tolerance of one another. The resulting liquid has the specified new reagent (or the unknown reagent) and the same volume as the original liquid. The volume of dried liquid specified is that of the liquid before it is dried. It may require use of dried liquid at more than one pad.

Level 5: The Recipe Level

At the recipe level, the operations involved are much the same as at the operation level, but the crucial difference is that instead of performing them one after the other, the steps of mixing, heating, buffering, etc.—and likely some notions of conditionality based on detectors—are used to create a *recipe*,[[25]](#footnote-25) which the program asks the board to perform:

RecipeResult perform(Recipe recipe)

List<RecipeResult> perform(List<Recipe> recipes)

The library can look at the various steps and the temporal constraints they imply and determine what steps can be performed in parallel. For example, if different liquids need to be heated, it may be possible to do that at the same time in different wells. Also, if there are several recipes to be performed, the system may determine that the steps may be interleaved or performed simultaneously. Finally, the recipe or recipes may be analyzed before being run:

RecipeAnalysis analyze(Recipe recipe)

RecipeAnalysis analyze(List<Recipe> recipes)

The analysis can be used to determine an optimal allocation of initial liquids to wells, possibly deciding to put the same reagent in multiple wells to allow for greater parallelism. It may also be used to drive a pipettor to realize that allocation.

Configuration

Before you can use a board, you need to get a reference to one. This is accomplished by calling

Board find\_board(String implementation\_name, [String instance\_name])

The implementation name is used to load (if necessary) a particular implementation (e.g., for a particular product). If the instance name is provided, a check is made as to whether a board with that name has already been prepared. If not, a new one is created and tied to the underlying implementation.

This approach allows a single program to control multiple boards and also allows the program to easily switch between running on actual and simulated hardware.[[26]](#footnote-26)

For things that might be part of the overall setup, but not necessarily provided by the manufacturer (e.g., cameras, pipettors), the user calls

Thing find\_thing(String implementation\_name, [String instance\_name])

And the board’s

void add\_thing(Thing thing)

This adds the new thing to the appropriate list on the board.

For detectors (whether hardware or software), the board similarly supports

void add\_detector(Detector detector)

Implementation

My original thought was to implement the library in C++ and put wrapper implementations on top of it for various superstrate languages. For various reasons, I’m now leaning toward implementing the first cut entirely in Python[[27]](#footnote-27) and having that implementation only supporting Python.

Platform implementation

Each platform implementation (e.g., OpenDrop, Joey on OpenDrop, Joey, Kanga), will be responsible for creating the board object and all of its component objects (wells, pads, etc.). In some cases, this will mean implementing subclasses for library-provided classes in order to perform required actions. In others, it will mean providing characterization parameters to constructors.

To actually communicate with the device, the board subclass will provide

def update\_state(self)

This does whatever is necessary to tell the device to change to the desired new state. It is up to the platform implementation to keep track of what that should be. Descriptions of desired changes are made via objects that support

def set\_state(self, new\_value)

taking new values of the appropriate type. Pads, heaters, and the like all provide a new\_state member of that supports this method.

update\_state() may also be asked to communicate with the device to read the state of a device. This is done by means of an object that supports

def read\_state(self, callback)

which instructs the board to acquire the state and then pass it to the provided callback object. Heaters, for example that can read their current temperature will provide a current\_state member of this type.

Platform example

As an example, consider OpenDrop, whose controller expects to receive exactly 128 bytes over its USB port. The board implementation could be something like

class Board(MPAM.Board):

def \_\_init\_\_(self):

self.next\_states = bytearray(128)

def update\_state(self):

// send the byte array to the device

The work of maintaining the array is done by the pad implementation, something like

class Pad(MPAM.Pad):

def \_\_init\_\_(self, x, y, board):

super().\_init\_\_(x, y)

self.index = board.max\_column\*y + x

self.state\_array = board.next\_states

self.new\_state = self

def set\_state(self, new\_state):

self.state\_array[self.index] = 1 if new\_state else 0

Library implementation

My current intention is to implement the library by using three worker threads and a ThreadPoolExecutor. The worker threads are

* the *device communication thread*, which has sole responsibility for communicating with the board and external devices,
* the *timed callbacks thread*, which is responsible for invoking callbacks after various delays, and
* the *system clock thread*, which is responsible for processing callbacks and updating state when the board’s clock is running.[[28]](#footnote-28)

Shared data structures

These threads (as well as the main thread, any explicitly created user threads, and threads in the thread pool) communicate by means of shared data structures. These data structures will be managed by Python synchronization constructs (probably largely condition variables), which make liberal use of locks. This goes against years of trying to use lock-free approaches wherever possible for performance, but (1) Python doesn’t really appear to have any support for lock-free programming and (2) my expectation is that these systems are going to be largely I/O-bound (and sleep-bound) anyway, so it probably isn’t worth trying to optimize.

The main shared data structures are

* the *communication request queue*, whose elements each contain
  + a list of *requests*, each of which is a call to set\_state() or read\_state(), closed with, respectively, the new state or callback,
  + a callback to invoke some time after update\_state() is called, and
  + an indication of when to invoke the callback, either a delay time, a wall-clock time, or a number of ticks.
  + an indication of whether the callback is “light” or “heavy”. Light callbacks are invoked directly, while heavy callbacks are submitted to the thread pool executor.
* the *timed callback queue*, a priority queue of callbacks sorted by desired invocation (wall-clock) time
* the *gated request operations queue*, containing elements to be put on the communication request queue during update\_state()
* the *pre-update callbacks queue*, containing callbacks to be invoked at the beginning of update\_state(), each accompanied by an integer tick delay.
* the *post-update callbacks queue*, containing callbacks to be invoked at the end of update\_state() after communicating with the device, each also accompanied by an integer tick delay.

The device communication thread

The device communication thread pulls items off the communication request queue and invokes them, retaining them on a local queue. When the queue is empty, it calls update\_state() and then schedules the callbacks on the retained items. Then it waits for the communication request queue to be non-empty.

The timed callbacks thread

The timed callbacks thread sleeps until the first item on the timed callback queue is due, interrupting its sleep if a new element is added. The callback is executed, and the return value is an indication of when (wall-clock or delay) to re-run it or None if the callback should not be repeated.

The system clock thread

The system clock waits for one of two events. If the clock is running, it sleeps until its next wall-clock time tick (interrupted if the running flag is changed to false). If the clock is not running, it waits on an event set by advance\_clock() (or when the running flag is set to true).

When it wakes up, it notes the time, then walks the pre-update callbacks queue. For each callback, if the delay number is greater than zero, it is decremented. Otherwise, the callback is invoked or scheduled.[[29]](#footnote-29) The return value is a new delay for the object or None if the callback is to be removed.

Next, it adds all of the items in the gated request operations queue to the communication request queue, preceded by one that does nothing in its request but sets an event in its callback. It waits on that event, signaling that update\_state() has completed.

Then it processes the post-update callbacks queue exactly as it did the pre-update callbacks queue.

Then it increments the current tick number.

Finally, if the clock is running, it notes that the next tick will come at the current interval time following the time it remembered when it last woke up and waits until that time.

[C++-based] Platform Implementation

The following sections on platform a library implementation, with a gray background, are my original implementation ideas, involving a C++ substrate library. This is probably the way to go long-term, but in the interest of getting a Python implementation up quickly, I’m currently planning on going with the pure Python implementation described above. I’m leaving the sections here as a reminder. [4/25/21]

My current thought is that a platform will be implemented by writing a small library in C++, and that this library will suffice for sticking under any language.

There will be classes mimicking most of the API classes,[[30]](#footnote-30) as well as for some things like dispensing sequences (which will likely actually be sequences that the API implementation will walk through).

For memory management purposes, the implementation classes will use shared and weak pointers.

The primary jobs of the implementation are

1. to list the components of the system (e.g., wells, chillers, magnets) and their associations to one another,
2. to provide characterization values for the board and the components, and
3. to actually interface with the underlying hardware.

The results of the first two will be used to create objects in the target language. The third will be used by the API implementation to do things like set electrode and magnet states and request heaters and chillers to aim for a particular temperature.

To handle this last capability, two abstract classes are provided:

class settable\_state<typename T> {

virtual void set(const T &value, bool is\_gated) = 0;

};

class readable\_state<typename T> {

virtual T read() = 0;

};

The board implementation will provide, e.g., something like

shared\_ptr<settable\_state<on\_off\_state>> pad\_state(int x, int y);

and it will be up to the implementation to return a subclass that knows how to actually turn the pad’s electrode on and off, either immediately or at the next clock tick. Similarly, a heater will provide something like

shared\_ptr<settable\_state<temperature>> requested\_temperature();

shared\_ptr<readable\_state<temperature>> current\_temperature();

The latter can be return a null value, indicating that the heater cannot monitor its current temperature.

Dealing with gated and immediate operations

To actually perform gated operations, the board provides

void process\_tick();

The board is responsible for remembering what gated operations need to be performed when this is called. For example, if I understand it correctly, OpenDrop boards set all their pad and well electrodes in a single call by passing in a 64-bit integer with one bit per electrode, 1 if the electrode is to be on and 0 otherwise. This could be implemented as something like:[[31]](#footnote-31)

class open\_drop\_board : public board

{

void set\_electrodes(uint64\_t states);

uint64\_t current\_states;

uint64\_t gated\_states;

uint64\_t gated\_mask;

void process\_tick() override {

current\_states &= ~gated\_mask;

current\_states |= gated\_states;

set\_electrodes(current\_states);

}

};

class pad : public settable\_state<on\_off\_state>

{

open\_drop\_board &\_board;

const uint64\_t my\_mask;

void modify\_states(uint64\_t &states, on\_off\_state s) {

if (s == on\_off\_state::on) {

states |= my\_mask;

} else {

states &= ~my\_mask;

}

}

void set(const on\_off\_state &s, bool gated) override {

if (gated) {

\_board.gated\_mask |= my\_mask

modify\_states(\_board.gated\_states, s);

} else {

modify\_states(\_board.current\_states, s);

\_board.set\_electrodes(\_board.current\_states);

}

}

};

[C++-based] Library Implementation

I’m currently waffling on how much of the actual library implementation should be done in the underlying C++ library and how much should be implemented in the target language. Arguments in favor of doing most of it “underneath” include

* C++ will almost certainly have better and more efficient support for concurrency than the target language, and other things (e.g., routing algorithms) are likely better done there.
* Anything that’s in the underlying library doesn’t have to be reimplemented for each language, making it easier to retarget the library to new languages.

Arguments against include

* There’s often a noticeable penalty when calling substrate code from other languages.
* It can be a real pain to call “up” from substrate code to superstrate languages, making callbacks difficult (and, in any case, target-language dependent).
* Superstrate languages typically have their own memory management, which can be tricky to get right from substrate code.[[32]](#footnote-32)
* Any standard concurrency/asynchrony model may not feel natural in some target languages.

Given time, my inclination would be to start with a C++ target—so there are no boundary issues to worry about—and let other target languages implementations layer on top of that and use as much or as little as they find useful.

Unfortunately, it doesn’t appear that C++ is a desired target, and we’re going to want to have at least the first few levels implemented in something[[33]](#footnote-33) relatively quickly, so it probably makes more sense to go with a minimal core in C++ and implement most of it in the superstrate.

1. I want to emphasize that this is my *current* thinking. Every time I talk to Viktor, I learn something that makes me change my understanding about how some part of it needs to work. [↑](#footnote-ref-1)
2. Not row and column, because people will use these in the wrong order when specifying coordinates. [↑](#footnote-ref-2)
3. I’ve been bitten enough times by passing in bare numbers to functions that assumed a different unit that I refuse to specify an API that lets it happen. I’ve written general packages for this in C++, Java, and Python, so I know it can be done. [↑](#footnote-ref-3)
4. I’m not really sold on this name. “Sample” seems to be used in a specific way in the field and would likely be confusing. “Fluid” is no better than “liquid” and seems even more to refer to the substance. Viktor uses “aliquot”, but I don’t know how transparent that would be. I could be persuaded to use “quantity”, but I tend to use that term for dimensioned types. “Quantity of fluid” is too verbose. [↑](#footnote-ref-4)
5. And I’m pretty sure all do by now. [↑](#footnote-ref-5)
6. It is possible that this will be dependent on drop size and/or presence of beads. If so, and this is knowable, we should make it specifiable. If it is unknowable, we should conservatively use the value for a maximal size drop with beads. [↑](#footnote-ref-6)
7. It’s possible that we will want to generalize this to a list of exit pads if we envision boards that have a single reservoir with multiple exits, which might make some operations work faster. [↑](#footnote-ref-7)
8. Okay, that’s confusing. My understanding is that Joey will gang together electrodes in several wells, and OpenDrop gangs together two electrodes in each well. Each such well and index will have its own well section (and, if the system models it, its own drop there), but changing the state of one well section will change the state of all well sections ganged with it. [↑](#footnote-ref-8)
9. The name notwithstanding, I don’t think that there’s any reason to expect to be able to actually see the sequence of actions (since they may be conditional on current state) so it may be worthwhile to find a different name. [↑](#footnote-ref-9)
10. Viktor calls this the “cooling rate”. I’m not sure that works for chillers. [↑](#footnote-ref-10)
11. If we assume that this follows Newton’s law of cooling, it could be specified by an exponential constant. [↑](#footnote-ref-11)
12. I’m assuming that it only makes sense for chillers to be associated with wells. [↑](#footnote-ref-12)
13. About 35 years ago, I was working on a project developing a programming language for wafer fabrication, and we caused a bit of a stir by proposing that “grad student” should be a subclass of “robot”, as their main value (from our point of view) was that they could carry boats of wafers from one device to another. [↑](#footnote-ref-13)
14. Okay, OpenDrop. And probably Joey. [↑](#footnote-ref-14)
15. There’s an interesting question of what should happen if you register a callback while the callbacks are being processed. That is, should they be registered relative to this clock tick or deferred to the next one. I’ll have to think about that some more. [↑](#footnote-ref-15)
16. I guess that it couldn’t hurt to have convenience wrapper functions for these cases, but it will mean that we’ll need to consider the error modes (e.g., what happens if you try to turn on the magnet at a pad that doesn’t have one). [↑](#footnote-ref-16)
17. I’m torn as to whether to throw an exception when dispensing onto a full exit pad. [↑](#footnote-ref-17)
18. I’m not sure exactly how this works when the pipettor is a human. [↑](#footnote-ref-18)
19. Or, perhaps better, once a drop moves onto the pad. [↑](#footnote-ref-19)
20. This parameter should almost certainly be a keyword parameter for languages that support them and should be assumed to be on any operation that has shuttling motion. [↑](#footnote-ref-20)
21. I’m not actually sure this is possible. [↑](#footnote-ref-21)
22. I’m currently going under the assumption that trains that don’t mix into other trains must end in wells. [↑](#footnote-ref-22)
23. What I’m thinking of is something like “no more than 5% difference”. [↑](#footnote-ref-23)
24. I’m probably going to want to relax this. [↑](#footnote-ref-24)
25. Yes, the details of how this is done are still fuzzy in my head at the moment. [↑](#footnote-ref-25)
26. Note to self: This isn’t quite right. If I want to be able to simulate arbitrary hardware, I still need a way to specify the configuration that I want to simulate. [↑](#footnote-ref-26)
27. Which will be interesting, as it will be my first attempt at anything non-trivial in Python. [↑](#footnote-ref-27)
28. This could probably be rolled into the timed callbacks thread, but it would need to submit the actual work to the thread pool, which doesn’t seem like a good idea. [↑](#footnote-ref-28)
29. I need to think through what happens with heavy callbacks. Do I wait for them to be done before doing the update? And how do I do the queue management if I don’t get a return right away. [↑](#footnote-ref-29)
30. In C++, these might *be* the API classes [↑](#footnote-ref-30)
31. Simplifying and ignoring atomicity. [↑](#footnote-ref-31)
32. For my last major project, we had no choice but to bite the bullet and develop a full-fledged garbage collector for C++. This made things *much* easier, but I don’t think I’d recommend it here. [↑](#footnote-ref-32)
33. Probably python, although I might lobby for Java. [↑](#footnote-ref-33)