A Sketch of a Multilevel API for Microfluidics Platforms

Evan Kirshenbaum

*Saturday, April 17, 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.

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 **up**, **down**, **left**, or **right**.[[2]](#footnote-2)
* ***XY coordinates*** have an integers *x* and *y*[[3]](#footnote-3) 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*.[[4]](#footnote-4) 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. 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*[[5]](#footnote-5) 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.
* ***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,[[6]](#footnote-6) 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.[[7]](#footnote-7) 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 *dead*. A dead 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 four *directions*. A neighbor will be a pad if the resulting coordinate is on the board and refers to a pad that is not dead. 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.

Wells

A well is characterized by

* Its *capacity* (a volume).
* Its *exit pad*. The location where drops from this well will be dispensed.[[8]](#footnote-8)
* 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.[[9]](#footnote-9) 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*.[[10]](#footnote-10) 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.
* A flag indicating whether the volume of the content is *exact* or *inexact*. If it is inexact, the volume measure will be taken to be a lower bound (which will never be allowed to be less than zero). This will be useful for the specification of wells asserted to contain an unknown (but sufficient) quantity of a given reagent.
* 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* (or *recovery time*). This characterizes the time taken for a heater to return to ambient temperature after it has been turned off.
* 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[[11]](#footnote-11)* 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 *region* that the thermometer reads from.

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.[[12]](#footnote-12)

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.
* 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:[[13]](#footnote-13)

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, wait the board’s pad motion time and return. 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.
* 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(Callback callback)

void after\_clock\_tick(Callback callback)

Finally, there are two methods that allow the program to pause until the next clock tick

void await\_clock\_tick()

and until the next clock tick when it appears that the system is idle[[16]](#footnote-16)

void await\_idle()

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. Or perhaps **north**, **south**, **east**, and **west**. This will work better if we think we might want to do things that involve caring about, e.g., a pad’s northwest neighbor. [↑](#footnote-ref-2)
3. Not row and column, because people will use these in the wrong order when specifying coordinates. [↑](#footnote-ref-3)
4. 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-4)
5. 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-5)
6. And I’m pretty sure all do by now. [↑](#footnote-ref-6)
7. 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-7)
8. 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-8)
9. 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-9)
10. 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-10)
11. I’m assuming that it only makes sense for chillers to be associated with wells. [↑](#footnote-ref-11)
12. 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-12)
13. Using Java-ish syntax, ’cause I have to choose some language. [↑](#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’m not sure exactly what that means, but roughly what I have in mind is when all multi-step gated operations have completed. Obviously this needs more thought. [↑](#footnote-ref-16)