WaveLight

The official specification for learning and understanding the thought framework.

By: Evan Smith

Table: Key Dictionary 

Field Name	⭐Key_ID	Key_Name	Alternative_Name	Specification	Definition
Data type	Integer	String	String	String	String

- The Key Dictionary table is used to store all of the WaveLight **keys** that are used to create **sequences**.

- The dictionary is pre-populated with all of the keys necessary to create WaveLight sequences.

- Here are the descriptions of the fields in this table:

  - Key_ID: A primary (unique) field that identifies any one given key.
  - Key_Name: The name for the key- sometimes the name does not effectively express the function of the key (e.g. what the key does), other times it does.
  - Alternative_Name: If a given key has a creative name that does not effectively express the function thereof, it will have an alternative name that is used to describe its function.
  - Specification: Used to categorize a given key based on functionality type.
  - Definition: The description of what the key does exactly.

Key-Calling Constraints

- Calling a key means utilizing its programmed logic (i.e. commanding it to begin its function) within a sequence (a sequence is a set of **called keys**) so as to generate a logical, compound function built from the micro-functions of the keys being used.

- Currently, WaveLight keys are called using a chording device that associates each unique key in the framework with its own preset chord (or button). Pressing a chord/ button associated with a WaveLight key is equivalent to calling said key.

- Chording devices (such as the Twiddler made by Tek Gear) are small, hand-held keyboards that can be used to type on various devices via Bluetooth.

- The Twiddler (shown to the left) and other chording devices are special keyboards in the sense that the keys are able to be programmed into chords which are specifically designated sets of keys pressed in unison to produce a different output than pressing the involved keys individually does.

- While typical keyboards have fixed buttons for letters like ‘a’ and ‘b’, these devices have the ability to combine keys and use the combination thereof to produce an entirely different output. For example, you could program two keys (out of the 20 total) on the Twiddler to output ‘a’ and ‘b’ respectively and then you could also program the combination of the two keys to output the letter ‘c’.

- A chord is pressed (e.g. on the Twiddler) by pressing all relevant keys which form the chord (at the same time or in any order) and reaching a state where they are all pressed down without having had stopped holding any previous key(s) (e.g. leading up to the last one) down, and then releasing one or more of the keys.

- A chording device is a good fit for calling WaveLight keys for a few reasons including the many possible chord combinations (thousands for the Twiddler) which are

necessary to include the vast number of WaveLight keys as well as the mobility of the device which allows the user to engage in the WaveLight sequence building process without having to look down at a screen or piece of paper.

❖ Key calling is used to form compound functions by calling the keys in a specific order. The following constraints (rules) apply to key calling in WaveLight:

→ The same key cannot technically be called twice in a row, but its function can be repeated (achieving a back-to-back effect) with a key that specifically duplicates keys' functions (which is defined on page X).

→ There are specialty keys that are used to disable **key components** (the pre-defined logic function that keys are given) which effectively removes the key from the sequence as if it were never called. Keys (i.e. components) can be re-enabled after being disabled and their function can still be used, although the time at which the disabling or re-enabling occurs can change the way that the key is applied.

→ For example, the function of keys that have been called will either be ongoing or complete. Disabling keys only applies to ones wherein the respective function has not completed. Keys that have already completed their function cannot be disabled.

Table: Called Keys 

Field Name	Called_Key_ID	Detected	Completion_Status	Enabled
Data Type	Integer	Date/Time	String	Boolean

- The Called Keys table is used to keep track of keys that have been called.

- Here are the field descriptions:

  - Called_Key_ID: A foreign (in database terms) field that references Key_ID from the Key Dictionary table.
  - Detected: The time at which a particular key was detected (called).
  - Completion_Status: The completion status of a key's function is either ongoing or complete.
  - Enabled: True if the key component of the called key is enabled.

Keyset: Capturing Keys 🔑

- Capturing keys are keys that 'point' to specific [1]events or moments within the sensory (or thought-based) input/ output stream so as to form a **capture**. Of the 13 capturing keys, eight of them are capable of performing their respective capture functions without the use of an already existing **timestamp** (a predefined moment between two points in time):

  - 🗝 Sound In Mind Capture: Used to capture sound spoken within the mind by the self.
  - 🗝 Sound In Real Life Capture: Used to capture sound spoken in real life created by anyone (including the self).
  - 🗝 Sound In Real Life From Self Capture: Used to capture sound spoken in real life created exclusively by the self.
  - 🗝 Memory From Mind Capture: Used to capture memories that formed as a result of a connection made solely within the mind (e.g. remembering a thought that was not shared with anyone else).
  - 🗝 Memory From Real Life Capture: Used to capture memories that formed as a result of something occurring in real life (e.g. remembering something someone actually said).
  - 🗝 Decision Capture: Used to capture a decision that has been contemplated but not yet decided upon. Namely, 'to get chocolate ice cream' and 'to get vanilla ice cream' are examples of decisions that could respectively be captured, whereas 'what ice cream to get' is not.
  - 🗝 Person Thought Capture: Used to capture the thought of a (real, human) person. This thought follows more technical constraints than the other ones listed on this page. For that reason, the next section (beginning on page 9) is dedicated to discussing the function thereof.
  - 🗝 Aura Sight Capture: Used to capture the artificial sight of an aura.

[1] The user may not achieve complete functionality without the use of a BCI or otherwise manual entry of timeline.

- The eight keys above are categorized as **standalone capturing keys** because they are able to be used without referencing a previously defined timestamp. That is, these capturing keys are able to function on their own by targeting what are known as **distinguishable events**.

- Distinguishable events are those which can be observed as unfolding in an individual (separate) and sequential manner within the sensory input/ output stream. For example, the three sound-based capturing keys listed above all default to capturing the most recent word that the user detected, the memory-based capturing keys default to the most recent memory, the decision key defaults to the most recent decision, etc. This default function will be referred to throughout the manual as the one-back default behavior. Channels that do not contain distinguishable events are ones wherein there is no 'break' in the sensory input/ output stream. For example, one cannot clearly distinguish one period of (real) sight from the next or from the previous.

- The five remaining capturing keys are ones which require a previously defined timestamp in order to execute their respective capture functions. Again, a timestamp is a period of time defined by two points in time. For example, time t1 = 0 seconds to (->) time t2 = 1 seconds creates a single timestamp.

- A timestamp is defined (created) in WaveLight by using the capturing keys listed above to capture two different events and then by linking the two with one another.

- For example, if the user said something (such as a word) out loud (e.g. at time t1 = 0 seconds) and then some time after that (at time t2 = 1 seconds), someone said something as well (another word), the user could create a timestamp between those two words by capturing them both and then linking them.

- The key used to link captures in WaveLight is called 'Moment Link'. The key is discussed in greater detail on page X where it is defined. For now, simply understand the general idea of what a timestamp is in the context of WaveLight: a byproduct of linking two separate captures together with one another.

- Here are the five remaining capturing keys and their respective capture functions:

  - Physical touch capture: Captures sense of physical touch.
  - Scent capture: Captures sense of smell.
  - Sight capture: Captures sense of sight.

- 🗝 Taste capture: Captures sense of taste.
- 🗝 Free-flowing thought capture: Captures thoughts within the mind not associated with other capturable items (all of which have been listed above) that are thought-based such as memories, decisions, and thoughts of people.

❖ The five keys listed immediately above get paired with a timestamp using a key called ‘Tower Refract’ (defined on page X) which detects an already existing timestamp and applies the specified capture key to said timestamp. This works by triggering the ‘Tower Refract’ key (like flipping a switch to on) then proceeding to call one of the capturing keys while the ‘Tower Refract’ switch is flipped.

❖ The result is a capture from within the specified **sensory channel** regarding the period of time defined by the timestamp. While the first eight (standalone) keys listed in this section are able to makes captures without using a timestamp, they too can be paired with timestamps using ‘Tower Refract’ to capture from within their respective sensory channel. Sometimes this is necessary even, such as when the sound the user wishes to capture is not a defined word. In that case, the user would need to define a timestamp (e.g. with the capture linking process) that covers the time wherein the relevant sound was produced, then capture it using the ‘Tower Refract’ key followed by the necessary capturing key.

Keyset: Person Thought Specific 🔑

- The ‘person thought capture’ key is unique in that it may require that the user specifies a particular person that they want to capture a previous thought of prior to making the capture. The user does this either by calling a new key (defined later in this section) and then continues to press a chord which is assigned to a specific person, or the other way is by [2]thinking about the specific person again between two new keys that enclose a specifier thought (a specifier thought in this context is just a subsequent thought of the same person who gets thought of specifically for the sake of capturing a previous thought of that same person).

  - 🗝 Person Thought Scan Start
  - 🗝 Person Thought Scan End

- The user thinks about the person that they want to target a previous thought of in between these two keys. The first instance of a thought of a person between these two keys does not count in terms of number notation for subsequent captures, but any additional thoughts thereof do. For example, if the user had thought some arbitrary set of people such as ‘Bob’, ‘John’, ‘Bob’, ‘Tom’, ‘Bob’, ‘John’ (where the respective repetition of names are only for single individuals (e.g. the first instance of ‘Bob’ is the same person as the second instance of ‘Bob’)) then immediately after that, the user called ‘person thought scan start’ and then thought of ‘Bob’ and then called ‘person thought scan end’ without having had thought of ‘Bob’ again (or any other person) prior to calling the ‘person thought scan end’ key, the most recent thought of ‘Bob’ (the instance thereof that was thought between the two scanning keys) does not count (for the sake of counting backward) until the ‘person thought capture’ key is called. Even if the user thinks of a different person and then that same person again (the latter being the one that the person thought register is set to) after having had set the register to that person, the instance of the thought of the person in between the scanning keys does not count for number notation until the register is reset. This is simply a particular constraint that makes capturing the right instance of a thought of someone more efficient and it also makes the process non-ambiguous regarding subsequent thoughts of people thoughts being targeted.

[2] The user may not achieve complete functionality without the use of a BCI or otherwise manual entry of timeline.

- At this point in the example, there would only be three instances of 'Bob' that have been thought which would be considered (for the sake of counting backward) for capturing. If the user proceeded to call 'person thought capture', that last instance thereof which was thought between the scan keys is directly used to capture a previous instance thereof and now gets considered in terms of number notation meaning that there are now four instances of the thought of 'Bob' which are up for capturing (when including the previous one that was just captured in this example- there would only be three after one gets moved). The purpose of discounting an instance of a thought of some specific person is mainly related to preventing the user from capturing that specific instance (e.g. the user likely does not intend to) but rather, a previous one (which is most often what use-cases involve).

- If we skip back a few steps in the example and instead of calling 'person thought scan end' after the most recent thought of 'Bob', the user had thought ,… 'John', 'Bob' (that is, if the user thought more than a single thought of 'Bob' after triggering the 'person thought scan start' key then went on to call 'person thought scan end' afterward, then all (e.g. both) instances of 'Bob' that were thought after calling 'person thought scan start' count in terms of number notation for capturing however, thoughts of others such as 'John' in this example only begin to count after there are two or more thoughts thereof or until the user thinks the thought of a different person afterward (e.g. after the thought of 'John', such as in this example where there was a proceeding thought of a different person), at which point all thoughts thereof count. The only way to utilize the technique that discounts the thought of a person in terms of number notation for capturing is to think about them last (e.g. without thinking about anybody else subsequently) and to think about them only one time and to do both of those things (last-and-only) in between calling 'person thought scan start' and 'person thought scan end'. If those constraints are not adhered to, no person thought is discounted in terms of number notation.

- For example, if 'Bob' was thought of immediately after 'person thought scan start' was called and then subsequently, 'John' was thought of immediately thereafter and then the user called 'person thought scan end', then the thought of 'Bob' does count in terms of number notation, but the thought of 'John' would not. Again, the only time a thought of a person gets discounted in this manner is when they are the most recent person thought and also, they have only been thought once since 'person thought scan start' was called. When 'person thought capture' is called, if there is a thought of people that could potentially be discounted (through the last-and-only constraint) but a situation where that process has not been finalized occurs (finalization of setting the person thought register and discounting a thought of a person additionally is achieved by calling the 'person thought scan end' key while

adhering to the two necessary constraints, that person thought does get considered in terms of number notation for capturing and the scanning keys' ability to block an instance of a person thought from being considered is reset when 'person thought scan start' key gets called again (regardless of the whether the 'person thought scan end' key has been called or not). In this sense, the proper use of the scanning keys is indeed effective for discounting a person thought in terms of number notation, however the main function is to set the person thought register. Unless the user thinks of a person in between calling 'person thought scan start' and 'person thought scan end', that register will not be set (the register always gets set to the person that the user thinks of immediately before the latter key gets called) and unless the user adheres to the last-and-only constraint, then the last person thought prior to calling 'person thought scan end' is considered regarding number notation for capturing (of course, the register can be set without the instance of the thought of the person that is used to set it being discounted). Regardless of these functions being achieved, when 'person thought capture' is called, all preceding person thoughts count regarding number notation for capturing (after it completes its function), and the scanning thoughts get reset as if they had never been called.

- Intuitively, only zero or (at most) one person thought can be (temporarily at that) discounted (blocked regarding number notation consideration for capturing) at a given time using these keys.

- The person thought register can be set in a different, much simpler way (without going through the process defined above) by simply using *counting number* keys (defined on page X) or the default one-back behavior of said capture key to simply count backward in thoughts of people, however this method of capturing an instance of a thought of a person does not pair with the person thought register. If the user has called 'person thought scan start' and then proceeded to call 'person thought capture' without having had called 'person thought scan end', all thoughts of people count regarding number notation for capturing. Any time the person thought register has been set using the scanning keys, calling 'person thought capture' overrides the default (one-back) capturing behavior and attempts to apply any coinciding *instance number* keys (e.g. that the user has called) which are defined on page X. If there are none of those, the capture key attempts to apply a default capturing behavior that targets instance-'i' (meaning the most recent instance) to capture the most recent (considered) instance of the thought of the person that the register is set to. This default behavior is a way of targeting the most recent instance (e.g. prior to the one used to set the person thought register) of a specific person without pairing the 'person thought capture' key with *instance numbers*. If for whatever reason, the 'person thought capture' key does not make a capture

targeting the person that the person thought register is set to if it is set when that key is called, then the person thought register gets reset just like it does when a capture is successfully made with that same capturing key being called. In this context, it is important to understand that if the user wishes to capture the thought of a person using *counting numbers*, if the person thought register is set, they must unset it by calling the 'person thought register reset' key because otherwise, when no *instance numbe*rs are paired with the register, the default behavior will be to capture instance-'i' of the person the register is set to. When the user resets the person thought register, all preceding thoughts of people count in terms of number notation for capturing.

🗝 Person Thought Register Reset

- In general, the person thought register is to be paired with *instance numbers*
- to capture a specific instance of a thought of a person. This allows the user to capture thoughts of people in a forward-sequential manner (or with respect to the most recent instance (notated as instance i') by using 'i'-related keys which are defined on page X, either way) the user will need to set the person thought register manually. The user may use the 'person thought scan start' and 'person thought scan end' keys and a thought of a particular person they wish to target a previous thought of to set the person thought register, or they may use a quicker and more efficient technique described below which prevents the need to use the scanning keys altogether.

- The user may also set the person thought register using a chord that has been preset to a specific person. While this intuitively requires the user to already have a preset chord for a specific person, the process of setting the person thought register this way is much quicker and simpler and use-case wise, it can save a lot of time. Typically, the user would only add chords for specific people which the user thinks of frequently, because those people thoughts are more likely to come up in the thought space of the user. The person thought register is also used for aura moving purposes and for that reason, having something of a contact list of other users (although chords are not limited to WaveLight users specifically, but allows the addition of non-user people too) can be useful. To set the person thought register to a specific person using a chord assigned to that person, the user will call the 'chorded person thought register set' key and then afterward, the user will press a chord which corresponds to a specific person, the same way that they would call a key. Doing so, the user has set the person thought register to that specific person. The process of setting the person thought register in this manner does not allows for discounting of thoughts of people in terms of number notation for capturing. That is, if the user

thinks about a person (e.g. ‘Bob) as they press the chord that corresponds to that person, they are technically creating another instance of the thought of that person.

🗝 Chorded Person Thought Register Set

Register: Person Thought 📋

Register name: Person Thought; Data type: String; Possible Values: NULL, *Names of people*

- A **register** in the WaveLight framework is a global variable which in computer science terms means a variable that all program functions have access to. When any given key is called, its program function has access to the entire list of registers and their states. The states of registers often affect keys' functions.

- The person thought register is used to keep track of which person the 'person thought capture' key will target when called in conjunction with *instance numbers* or otherwise, how it will function.

- The 'person thought capture' key can pair with multiple number keys and can make multiple captures with a single call thereto. *Counting number* keys and instance number keys can be paired with the capture key, but not both at once. Whichever type of number key (of the two that the capturing thought can be applied with) is called first in time will be the type which multiple captures can occur with regard to.

Capturing Constraints 

- At any point in time, a given time period (e.g. moment) within a particular capturable channel (think: there is one channel per capturing key) from the sensory input/ output stream can either be non-captured, captured, or captured and moved into an aura. Moving a capture into an aura is the same as the term to **channel** the capture.

- For a given time period occurring between two points in time (e.g. between points ‘X’ and ‘Z’), unless the period within a particular channel is captured and channeled, the points included in the time period can continue to be referenced even if one or both are already involved in a capture.

- This means that if there exists a **timeline** of three points such as ‘X’, ‘Y’, and ‘Z’, and capture ‘A’ between points ‘X’ and ‘Z’ has been made already but the capture has not been channeled into an aura, it is still possible to individually reference points ‘X’ and ‘Z’ both so as to create (a) new capture(s) related to one or both of those points.

- Referencing points from previous captures is useful when the user wishes to specifically move (e.g. channel) a portion of a capture into an aura rather than the whole capture. For example, the user might originally capture from the sensory input/ output stream between points ‘X’ and ‘Z’ (from the previously defined timeline) and then later decide that they only want to channel the period of time between points ‘X’ and ‘Y’.

- One way to isolate a portion of an already existing capture is to effectively separate it into a new capture. From the example above, the user could go through the capturing process again, this time targeting the period of time between points ‘X’ and ‘Y’ to form capture ‘B’. In this situation, there would exist two separate captures (one between points ‘X’ and ‘Z’ (capture ‘A’), and a second (capture ‘B’) between points ‘X’ and ‘Y’). From there, the user could move capture ‘B’ (that targets the period of time between points ‘X’ and ‘Y’) on its own.

- In a different scenario using a new timeline of points: ‘D’, ‘E’, ‘F’, ‘G’, if a capture between points ‘E’ and ‘F’ was made to begin with as capture ‘C’ and the capture was channeled into an aura and then a second capture ‘W’ was made between points ‘D’ and ‘G’, the latter capture would not include anything from points ‘E’ through ‘F’ (inclusively) within its content. Captures that are channeled already are

thought of as never having had occurred within the timeline of events for the sake of future capturing. That is, as captures are made and channeled from the WaveLight timeline, the timeline itself begins to exclude those events as if they never occurred and thus, an altered timeline emerges. If capture 'C' had been channeled into an aura and then was later derived (i.e. removed from said aura) and placed back into the default storage location (aura derivation is described on page X) prior to capture 'W' being made, then the process of referencing and/ or capturing regarding the content included in capture 'C' could begin again as if capture 'C' were never channeled into an aura. That is, if capture 'C' was channeled into an aura and still resided there, then the timeline goes from point 'D' to point 'G' however if capture 'C' was channeled into an aura and then derived therefrom, then the timeline gets restored to 'D', 'E', 'F', 'G'.

❖ Capturing that is at all related to (i.e. overlapping regarding the content of) previous captures follows the following constraints:

→ If a single later (i.e. occurring later in time) capture includes the entirety of a previous capture or captures and additional content, the later capture overwrites (e.g. replaces) the previous capture(s) so as to include (e.g. merge) everything from the relevant previous capture(s) with everything from the most recent (latest) capture as one single capture. This feature can be disabled using a key (defined on page X) that prevents merging related to consecutive captures which in turn allows the user to capture only non-captured portions from the sensory input/ output stream while targeting a period of time which includes some percentage that has already been captured.

→ If a single later capture includes the entirety of a previous capture and only that (i.e. if the captures are exactly the same), no additional capture is made.

→ If a single later capture includes less than the entirety of a previous capture and no additional content (i.e. no content that the previous capture does not include), a new capture is made that includes only the content of the later capture (including the overlap) and the previous capture is left alone until one or both of the two captures are channeled at which point, the content of the individual captures can fluctuate. The rules regarding what happens exactly when captures overlap this way then subsequently get channeled is described in the next section of this manual on page X).

→ If a single later capture includes less than the entirety of a previous capture and additional content that the previous capture does not include, the same rule

applies: a new capture is made that includes only the content of the later capture and the previous capture is left alone until one or both of the two captures are channeled.

Table: Bubble of Captured Things 

Field Name	⭐Capture_ID	Marked	Called_Capturing_Key	Content	Began	Ended	Entered	Exited	Nullified	Partial
Data Type	Integer	Boolean	Integer	String	D/T	D/T	D/T	D/T	Boolean	Boolean

- The Bubble Of Captured Things (BOCT) table is where all captures are originally stored.

- All types of captures that are made using the capturing keys are stored in this table. If a capture or group thereof are channeled into an aura or auras then later derived from said aura or auras, the capture(s) re-enter the BOCT with a new ID.

- Here are the descriptions of the fields in this table:

  - Capture_ID: A primary field binding a unique ID to a particular capture.
  - Marked: This relates to a process known as *marking* which will be discussed later on in this manual beginning on page X. For now, just know that a particular capture can be marked (which would set this field to TRUE) or unmarked (as a capture is by default, making this field FALSE).
  - Called_Capturing_Key: A foreign field that points to the Called_Key_ID in the Dictionary of Keys table.
  - Content: The content of a capture is literally what was sensed by the user making the capture as expressed through the sensory channel it was derived from.
    - The format of content section is as follows: Words transcribed with CC style descriptions if necessary (for sound), *Memory of '…'*, *Decision to '…'*, *Thought of: '…'* (for person thought capture), and *Imagined sight of ColorChannel_AuraCode* (for auras). With more advanced BCI, the non-distinguishable-event-based capturing keys may have descriptions for their content as well (e.g. feeling of, smell of, sight of, etc.).
    - For now, non-transcribable captures (including touch, smell, sight, taste, and free-flowing thought experienced within a certain timeframe) will be null for this column however, there will still be the option for the user to manually enter capture content descriptions when working with the *WaveLight*

Simulator- a software being designed for the purpose of computationally deciphering WaveLight sequences according to the logic function of the keys.

- Began: When the sensory input/ output began (D/T is short for Date/Time).
- Ended: When the sensory input/ output ended.
- Entered: When the capture entered the BOCT table.
- Exited: When (if applicable) the capture exited the BOCT table.
- Nullified: If capture 'A' is replaced by a second capture 'B' because the content of 'A' is a subset of the content of 'B' then capture 'A' would get nullified or effectively deleted. This does not apply in a vice-versa sense. That is, if capture 'C' is already made and then capture 'D' is made and the content of capture
- 'D' is a subset of the content of capture 'C', capture 'D' does not immediately get nullified. However, if capture 'C' which is a superset of capture 'D' gets channeled, capture 'D' does get nullified or effectively deleted.
- Partial: If capture 'E' is made by targeting a time period between points 'X' and 'Z' in a timeline that consists of points 'W', 'X', 'Y', and 'Z' (in that order) and then capture 'F' is made by targeting the time period between points 'W' and 'Y', then captures 'E' and 'F' are both considered partial meaning that part (or all) of their content is referenced by another capture and therefore, the first capture to be channeled will assume the overlapping section (between points 'X' and 'Y' in this case) and if the other capture is channeled while the formerly channeled one still resides in an aura, the latter channeled capture does not include the overlapping section. If both captures are channeled at the same time, the latter capture ('F' in this case) would assume the entirety of the content it consists of including the overlapping section which would be subtracted from the content of capture 'E' by altering either the began or ended fields or possibly both.

Table: Superset Captures 

Field Name	Superset_Capture_ID	Subset_Capture_ID	⭐Exclusive_Portion_ID
Data Type	Integer	Integer	Integer

- The Superset Captures table is used to keep track of captures from the BOCT table which have the Partial field set to TRUE.

- This table is used to keep track of which captures are superset captures relative to other subset captures but in particular, which superset captures have portions that may be claimed by (an)other capture(s) should the other(s) get channeled first.

- Here are the descriptions of the fields in this table:

  - Superset_Capture_ID: References Capture_ID from the BOCT table to denote a capture that is a superset of another capture.
  - Subset_Capture_ID: References Capture_ID from the BOCT table to denote a capture that is a subset of another capture.
  - Exclusive_Portion_ID: A primary field which keeps track of the exclusive (non-shared) portions of a superset capture regarding a superset-subset relationship (which each row in this table represents).

Table: Exclusive Portions 

Field Name	Portion_ID	Start	End
Data Type	Integer	Date/Time	Date/Time

- The Exclusive Portions table is used to keep track of portions of superset captures from the BOCT table which are exclusive (non-shared).

- Here are the descriptions of the fields in this table:

  - Portion_ID: References Exclusive_Portion_ID from the Superset Captures table in order to define a particular exclusive portion of a superset capture.
  - Start: The start time for the portion.
  - End: The end time for the portion.

- In the event that a superset capture is broken into two or more portions due to a subset capture being channeled into an aura, the superset capture becomes multiple captures within the BOCT table- all with a new Capture_ID. If the subset capture that split the superset capture to begin with re-enters the BOCT table, as long as the portions of the original superset capture have not been separated (i.e. unless a portion but not the entirety of the original superset capture has been channeled) previously through one or more less than the total amount of portions having had been channeled either into different auras or at different times (and then derived- so that it currently resides within the BOCT), then the portions that the original superset capture was split into become nullified and the original superset capture which was previously nullified becomes unnullified (e.g. restored in this context). If from there, the subset capture is once again channeled, then the process reverses itself such that the portions that were nullified get unnullified and the original superset capture gets nullified (unrestored).

- If a superset capture is channeled at the same time as a subset capture thereof such that the subset capture assumes the shared portion when it is channeled into an aura, the split superset capture is considered as one single capture within that aura when it comes to aura deriving purposes (aura deriving is discussed later in the

manual, beginning on page X). That is, if a superset capture ‘X’ between points ‘A’ and ‘D’ in a timeline consisting of points ‘A’, ‘B’, ‘C’, and ‘D’ is channeled at the same time as a subset capture ‘Y’ between points ‘B’ and ‘C’ from that same timeline and during that process, capture ‘Y’ assumes the shared portion (between points ‘B’ and ‘C’) because it gets channeled first and therefore, the portions of capture ‘X’ that eventually enter the aura that the superset is channeled into include only points ‘A’ (inclusively) up to ‘B’ (non-inclusively) and ‘C’ (non-inclusively) up to ‘D’ (inclusively) then the user should understand that if they wish to derive from the aura that the portions of capture ‘X’ entered, then they should consider the portions of the original superset capture that entered the aura as a single capture and therefore, the portions can only be derived altogether (i.e. all at once). After the portions are derived in this manner, they are stored as separate captures within the BOCT table and moving forward, must be channeled and derived separately.

Keyset: Aura creation keys 🔑

- Aura creation keys are the keys that are used to create the aura containers that contain captures which have been channeled.
- The process of creating auras is very straightforward. Simply calling the relevant aura-creating key creates an aura of the specified color. After that, the created aura is ready to use.
- There are eight aura creation keys (one for each aura color):
  - 🗝 Blue aura creator: Creates the aura that accepts sound and *touch secondarily*.
  - 🗝 Green aura creator: Creates the aura that accepts touch.
  - 🗝 Orange aura creator: Creates the aura that accepts scent.
  - 🗝 Red aura creator: Creates the aura that accepts sight and *touch secondarily*.
  - 🗝 Yellow aura creator: Creates the aura that accepts taste.
  - 🗝 Purple aura creator: Creates the aura that accepts thoughts of people, free-flowing thought, and *sight secondarily*.
  - 🗝 Pink aura creator: Creates the aura that accepts memories, decisions, and sound with no primary.
  - 🗝 Cyan aura creator: Creates the aura that accepts sight and sound with no primary.

Aura Creation Constraints 

❖ An aura of a particular color can only be created if there exists one or more captures within the BOCT that qualify the creation thereof. A given capture can qualify an aura of a particular color to be created if it meets the following criteria:

→ The capture can be channeled into that type (color) of aura via the aura's primary or open (for pink and cyan auras) acceptance.

→ The capture is actively residing within the BOCT rather than contained in another aura.

→ There are no auras of the same color already created that the capture can enter.

❖ Basically, what this means is that an aura can only be created if there is a possible use case for it that a different aura could not already achieve.

❖ Thus, same colored aura cannot be created twice in a row using the 'duplicate' key. Any time an *aura creating* key is called and is not qualified to create an aura, it is permanently deleted.

Table: Auras 

Field Name	⭐Aura_ID	Called_Creator_Key_ID	Status	Counting_Code	Instance_Code
Data Type	Integer	Integer	Integer	Integer	Integer

- The Auras table is used to track the existence and availability status of auras at a given point in time.

- A unique ID is assigned to every aura that is created.

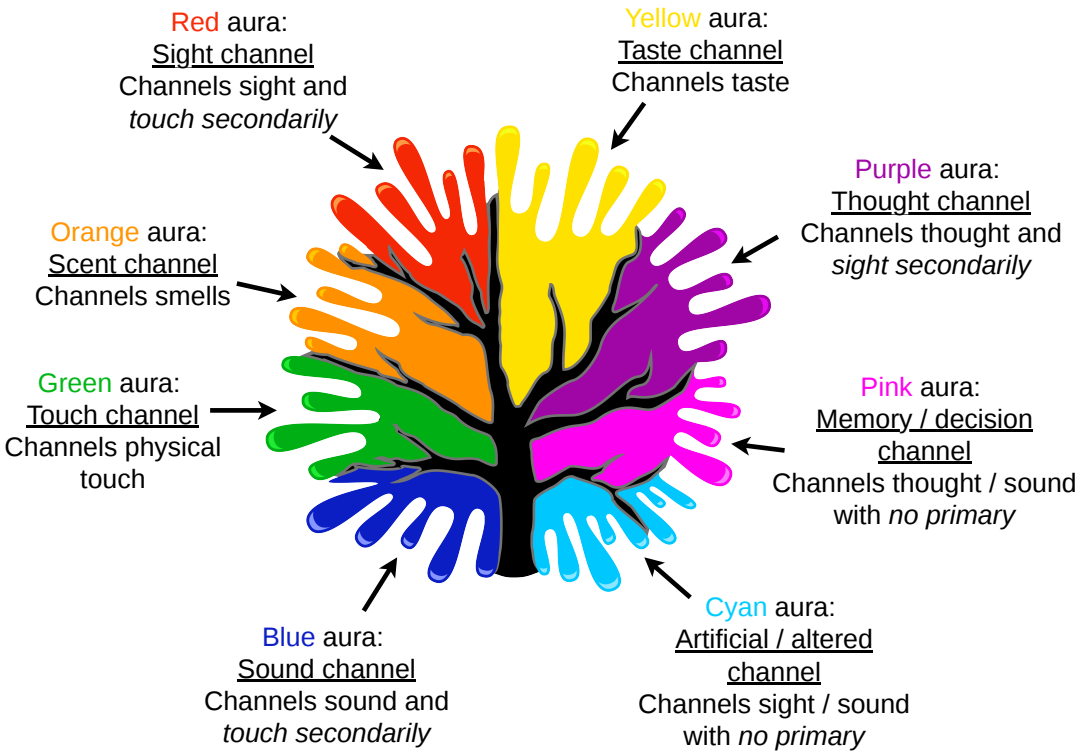
- Here are the descriptions of the fields in this table:

  - Aura_ID: The unique ID for a given aura.
  - Called_Creator_Key_ID: References Called_Key_ID from the Called Keys table.
  - Status: An aura's availability status is determined by a specifier label. The exhaustive list of specifier labels is listed below.
  - Counting_Code: For referencing purposes (defined on page X), the dynamic code used to reference an aura via the 'counting' reference technique.
  - Instance_Code: For referencing purposes (defined on page X), the dynamic code used to reference an aura via the 'instance' reference technique.

- Status labels for an aura (the integer is what will be stored in the table):

  - **Free (0)**: Aura is in its original location (non-moved) and it is non-containing.
  - **Containing (1)**: Aura is non-moved and contains some captured item(s).
  - **Activated (2)**: Aura is moved and the BOCT contains one or more captures that could be directly channeled into it aura via a private BOCT-driving key (see page X).
  - **Moved (3)**: Aura is moved, non-containing, and non-activated.
  - **Closed (4)**: Aura was in a containing state while moved and thus, cannot be utilized any further.

Diagram: Aura Acceptance 📋



- To channel a capture means to move it into an aura's container.
- For an aura to channel a sensory type *secondarily* means that the aura can accept that type of sensory capture as long as any such capture is accompanied by an instance of a different capture from the channel that the aura accepts primarily.
- For example, the blue aura accepts sound primarily, meaning that a single sound-based capture or several sound-based captures can enter the aura without any other type of capture being present. The blue aura accepts physical touch-based captures secondarily, meaning that as long as one or more sound-based captures accompany the touch-based capture(s), the touch-based capture(s) can also enter the aura.

Aura Acceptance Constraints 

- Auras in WaveLight are essentially the containers that are used to store the already discussed captures that come from the sensory input/ output stream.

- An aura can only accept captures or portions thereof that occurred in time (but were not necessarily captured) prior to the point in time at which the aura was created. For example, if a timeline of events in time included points ‘W’, ‘X’, ‘Y’, and ‘Z’ occurring in that sequential order and aura ‘A’ was created at the same time that event ‘Y’ ended, events ‘W’-‘Y’ could be channeled into aura ‘A’ (even if aura ‘A’ was created prior to those events being captured) whereas event (e.g. capture) ‘Z’ could not be contained by ‘A’.

- Each of the eight auras has a light and a dark side regarding the containing capability thereof. The light side of an aura is used to project or bring attention to a specific moment or set of moments (namely, a capture or a set of captures) from the sensory input/ output stream whereas the dark side of the aura is used to contain or take attention away from the capture(s) it contains.

- Every one of the auras in WaveLight follows the following constraints pertaining to storage capability:

  → An aura may accept a capture or a group thereof into its light side or its dark side two times per side. When multiple captures enter an aura at one time, this counts as a single acceptance from the aura.

  → An aura can be derived from (i.e. an aura’s contents can be removed from it and moved back into the default BOCT table storage space) as many times as the user wishes (e.g. until it is empty).

  → A given aura must be completely empty (e.g. non-containing) for it to accept a capture or group thereof into either of its sides.

  → No aura may contain captures within its light and dark sides simultaneously.

Keyset: BOCT driving 🔑

- The *BOCT driving* keys are used to move or 'drive' the BOCT table (which can be envisioned as a 'bubble' containing sensory captures and will sometimes be referred to as 'the BOCT' rather than 'the BOCT table' throughout the manual) through an aura or auras in order to channel the contents within the BOCT into said aura(s).

- The *BOCT driving* keys are not capture specific, meaning that they move the entire BOCT through the aura(s) being referenced and whatever the aura(s) can accept from the BOCT given the aura acceptance constraints, they will.

- For example, if the BOCT table had three captures in it including a touch capture, a sound-based capture, and a memory capture, using the same *BOCT driving* keys to drive the BOCT through different types of auras would play out differently in each scenario (assuming the respective auras were each created after all of the captures were made).

  - Driving the BOCT through a green aura would move only the touch-based capture out of the BOCT and into the green aura.

  - Driving the BOCT through a blue aura would move the sound-based as well as the touch-based captures out of the BOCT and into the blue aura.

  - Driving the BOCT through a pink aura would move the memory as well as the sound-based captures out of the BOCT and into the pink aura.

- These different outcomes could all result from using the same key when the BOCT table contains different things. What a given BOCT driving key does simply depends on what resides within the BOCT (capture wise) as well as what aura(s) is/ are driven through. Again, any eligible capture regarding the acceptance of a particular aura that the BOCT is driven through will leave the BOCT and move into the aura (with the exception of specifying that the capture should not, which can be done using a technique called *marking* that is discussed on page X).

- Four out of the six *BOCT driving* keys are known as public (versus private). Public *BOCT driving* keys are used to move the BOCT through auras that are not already moved. By default, an aura is non-moved and it remains that way until the user

moves it using *aura moving* keys as described on page X). The four public *BOCT driving* keys are:

- 🗝 Bright Passage (public show number one): The public show *BOCT driving* keys are used to drive the BOCT through the light side of non-moved auras. Any eligible capture(s) will enter the light of the aura(s). This type of key must precede the reference of the aura(s) it targets (aura referencing is discussed on page X).

- 🗝 Luminous Step (public show number two): ‘ ‘

- 🗝 Obscured Path (public hide number one): The public hide *BOCT driving* keys are used to move the BOCT through the dark side of non-moved auras. Any eligible capture(s) will enter the dark of the aura(s). This type of key must also precede the reference of the aura(s) it targets.

- 🗝 Skipping Stone (public hide number two): ‘ ‘

- ❖ The two private BOCT driving keys are alternatively used to drive the BOCT through already moved auras. When the BOCT is driven through a moved aura and a single capture (or set thereof) enters the moved aura in this situation, that aura becomes closed and can no longer accept captures, be moved, or be derived from.

  - 🗝 Illuminate Flow (private show): The ‘private show’ *BOCT driving* key is used to drive the BOCT through the light side of moved, **activated** auras. Any eligible capture(s) will enter the light of the aura(s). This key must proceed the relevant capture(s) being made, which must precede the moving of the aura.

  - 🗝 Shadow Current (private hide number one): The ‘private hide’ *BOCT driving* keys is used to drive the BOCT through the dark side of moved, **activated** auras. Any eligible capture(s) will enter the dark of the aura(s). This key must also proceed the relevant capture(s) being made, and that process must precede the moving of the aura.

- ❖ There are two more keys that will be introduced with this set because they are both related to the *BOCT driving* keys although the second one is not technically exclusively related to this set as it works with another keyset in the framework. Because this is the first time these keys come up, they will be introduced now. The

second key will also be mentioned again later with the other keysets it operates with.

- Capture break: The ‘capture break’ key is used with both the private and the public *BOCT driving* keys to split captures according to aura acceptance constraints. Sometimes, there are situations where the entirety of a capture cannot be channeled into a specific aura because of when it occurred within the sensory input/ output stream, yet part of it (e.g. some fraction of it) can be accepted by that same aura. For that reason, this key is available as a quick way for the user to split a capture or captures between what will go into an aura and what will not without having to go back and manually capture anything again. The key automatically breaks the capture into portions and puts what will go into an aura into it and what will not go into the aura back into the BOCT table with a new capture ID.

- Polarity override: The ‘polarity override’ key is used with the public *BOCT driving* keys to enable the user to expand the use capabilities thereof, as further discussed in the next section.

BOCT Driving Constraints 

❖ The *BOCT driving* keys operate in a complex way, although their use cases are typically quite straightforward. In the last section, the idea of 'any eligible capture' (in terms of what will be accepted by an aura when the BOCT is driven through it) was introduced. There are several additional constraints that are required to be adhered to for a capture to actually be eligible to be moved into an aura:

→ A given (public) *BOCT driving* key can only move a given capture one time. If a particular (public- because private movement is final) *BOCT driving* key has already moved a given capture, the exact key cannot be used to move that same capture again, although this rule does not constrain other captures from being moved by the key when it is detected even if the ineligible capture still resides within the BOCT. For example, if the BOCT had two captures (each of which were sound-based) and 'public show number one' had already channeled one of the two captures into an aura and then the same capture was derived from said aura (which would automatically move it back into the BOCT) and then later on, the 'public show number one' key was called again (assuming channeling both captures into a given aura would otherwise be eligible had 'public show number one' not already moved the one), the other capture (which was not already moved once) would move into the aura while the other capture (which had already been moved) would not .

→ The public *BOCT driving* keys operate on a polarity-alternating basis meaning that all four of the two types of keys (the two public show keys and the two public hide keys) require alternating between the two identically functional keys in the pair for a specific type so long as the polarity is not reset which occurs when the BOCT table is **prime** (the BOCT is prime when it has gone from containing captures to being empty and has not yet contained another capture since). This constraint applies even when a called public *BOCT Driving* key does not move any captures into an aura (e.g. all that needs to occur for the polarity to flip is a key must be called). Polarity is like an 'up next' variable that specifies which one of the two keys regarding the two pairs needs to be used next. The main purpose of the constraint is to guide a natural flow of key-calling with regard to keys like these which have a twin (e.g. equivalent) key for the sake of considering the constraint that limits a given one of these keys to moving a particular capture only one time. For example, if a particular movement key (such as one of the public show keys) had been used recently and the BOCT table

had not become prime since then, the functionality of either key of that type (e.g. 'public show number one' as functionally equivalent to 'public show number two') could not be utilized unless the other key with the same functionality gets used subsequently or if the polarity has been manually overrode. If the polarity is not right, the key does nothing even if the BOCT table contains captures that would otherwise be channeled.

→ The polarity of the public BOCT driving keys is manually changed using the 'polarity override' key that was introduced in the last section on page 30. This key is used with the public BOCT driving keys as well as several other types of keys (as aforementioned; that also operate on a polarity-based rhythm) to deliberately override the polarity constraint regarding the key type it gets used with.

→ The pairing of the 'polarity override' key with public *BOCT driving* keys (assuming the BOCT is not prime and that one of the public *BOCT driving* keys has been called since the BOCT became not prime) involves the 'polarity override' key being called in tandem with the same public *BOCT driving* key that was last used and before the other (equivalent functionality wise) public *BOCT driving* key is called. This may seem very tricky, but it is quite straightforward in practice. Here is an example:

- Let's say a capture was made and then an aura was created and the capture was channeled into the aura with 'public show one' (this sequence as a whole would require aura referencing which has not been discussed yet, but for the sake of this example it is safe to imagine). Immediately after that, a second capture is made and then a second aura is created. According to the polarity constraint, 'public show one' is not eligible to channel the second capture into the second aura yet because (similar to a possession arrow in sports) the polarity constraint requires alternation between the two public show *BOCT driving* keys (e.g. in this example) in between use cases within periods wherein the BOCT has not been reset (i.e. made prime). The point of the 'polarity override' key is to override this rule. In this case, calling the override key between calling 'public show one' (for the second time) and referencing the second aura would allow 'public show one' to channel the second capture into the second aura immediately without ever needing to call 'public show two'.

→ While the key-calling-polarity constraint has a work-around override, the single-move constraint does not, and it always stands. Consider the same example from right above. If everything was the same except instead of the second capture being different from the first, the second capture from the example was instead a derivation of the first capture from the first aura (that is, if there was only one

capture made which was subsequently channeled and then derived) and the user tried to move that first capture into the second aura by using 'public show one' for a second time along with overriding the polarity of the public show *BOCT driving* keys (assuming the key-calling-polarity constraint required such an override), the BOCT itself would move through the second aura (and if anything resided in the BOCT which was eligible to enter the second aura (e.g. other than that capture or which had not already been moved by 'public show one'), it would enter the aura) however the capture which had already been moved by 'public show one' would not leave the BOCT and would not enter the second aura.

- Again, the key-calling-polarity constraint is BOCT-specific and prevents the BOCT from moving at all if not adhered to. The single-move constraint is capture-specific meaning it does not prevent the BOCT from moving, but it filters out what captures within it can leave it and enter (an) aura(s).

→ If a capture is **set** meaning that there is a corresponding [3]activated aura ready to accept that capture with a single private *BOCT driving* key, then that particular capture requires the use of two public *BOCT driving* keys of the same type (e.g. both public show keys or both public hide keys) in order to override what is known as the lock-in rule which prevents a single public *BOCT driving* key from moving the capture in this situation (i.e. when it is set). The single-move constraint defined above still applies even when using the lock-in rule override technique. For example, if both public *BOCT driving* keys have already moved a specific capture, neither one is able to do so again. The key-calling-polarity constraint still counts but it is sort of automatically overridden when using the lock-in rule work-around technique because when the user calls both (i.e. the two) public *BOCT driving* keys of the same type and then references an aura, the way that the polarity is set beforehand does not matter. The BOCT gets moved according to the functionality of the relevant pair of keys (i.e. through the light or the dark of the aura that gets referenced) and then the polarity gets switched (e.g. alternates) to the complement (like a possession arrow getting flipped) that the polarity was not set to before the movement. If the key-calling-polarity constraint is not relevant in a scenario like this, the first of the public *BOCT driving* keys to be called is considered the one that moved the capture(s) and otherwise (i.e. if the key-calling-polarity constraint is relevant to the situation), the key that allows adherence to the constraint (e.g. the non-most-recently-used one) is considered the one that moved the capture(s) (for the sake of adhering to the single-move constraint moving forward). It is important to be considerate when using this technique because there may be a situation where a capture is set and the user wishes to channel a different (non-set) capture into a

[3] **Activated** is a term defined within the body of the manual on page 25.

different, non-activated aura. In that situation, the user only needs to consider whether the key-calling-polarity constraint regarding the public *BOCT driving* key type they wish to use is relevant because non-set captures do not get 'locked in' the BOCT like set ones do meaning that a single public BOCT driving key can still move the BOCT and channel non-set captures even when the BOCT contains set captures and prevents the latter of the two kinds from moving, although the set captures within it will not exit. The lock-in constraint is capture-specific, not BOCT-specific.

→ A capture does not require this sort of override (i.e. it does not become set) if it is captured after an aura becomes activated, even if it is otherwise eligible to enter that aura. Only captures that reside within the BOCT prior to an aura being moved can possibly enter that aura with a single private *BOCT driving* key. However, an aura that is moved upon a capture being made which prevents said capture from entering it for that reason can be moved back and afterward, the capture becomes eligible to enter the aura.

→ Again, if a private *BOCT driving* key moves or transfers (a) capture(s) into an aura (assuming the capture adheres to the constraints above and is actually moved), that key finalizes the state of the aura meaning that after that *BOCT driving* key completes the entirety of its function (which may move more than one capture into said aura), the aura is [4]closed.

→ If multiple auras are activated and there exists (a) capture(s) within the BOCT table which is/ are eligible to enter into more than one of the activated auras, a single private *BOCT driving* key called causes the BOCT to drive through all activated auras, channeling the respective eligible capture(s) into the light or dark (depending on the key type) of all of the activated auras by moving as many of the captures (if there are more than one) into the auras as possible (one at a time, based on the creation time of the auras, rather than the time at which the auras were moved). In a situation where *marking* of the BOCT captures is involved, the single private *BOCT driving* key that gets called drives the BOCT through all of the activated auras and only considers the captures within the BOCT that are marked for the entirety of its (driving) function.

→ The BOCT can only be driven through moved auras by private *BOCT driving* keys. Alternatively, public *BOCT driving* keys can only be used to move the BOCT through non-moved auras. If an aura is moved but it is not closed, the aura still counts in terms of the number notation for auras that have been created (number notation is

[4] **Closed** is a term defined within the body of the manual on page 25.

necessary for the sake of disambiguating the logic involved in *aura referencing* which is described on page X) but if an aura is closed, it is treated as if it were never created in terms of aura referencing regarding the number thereof.

→ It is possible for the user to call multiple public *BOCT driving* keys prior to applying any of them via an aura reference. In this situation (regarding situations involving any number of called public *BOCT driving* keys including at least one that is different from the rest and involving one or both types thereof), the first to be called of the remaining (e.g. which have not been disabled key component wise) public *BOCT driving* keys is the one that gets applied and the rest of the keys that were called after the one that gets applied but before said key was actually applied get deleted and never used. If the user calls a public *BOCT driving* key, disables it, and calls a subsequent *BOCT driving* key (including that same key) later on, they may re-enable that key and apply it so long as no more than one aura reference has been made while that particular key was disabled. That is, if the key that was called and then disabled is disabled during two instances wherein auras are referenced, that particular key gets permanently deleted.

→ The 'capture break' key is paired with the public and private *BOCT driving* keys in different ways. With private *BOCT driving* keys, the 'capture break' key works by breaking captures within the BOCT in such a way that among all the auras that the private *BOCT driving* key will drive the BOCT through, as much as the first aura (first in terms of creation time, rather than movement time) can accept (not just in terms of whole captures, but also including portions thereof) is divvied into the first aura driven through, then as much as the second aura can accept is divvied into it, and so on. When using the public *BOCT driving* keys with the 'capture break' key, at most one aura can be driven through at once, so this key simply breaks relevant captures into two, separating what can enter the aura from what cannot. Any time a remainder of (a) capture(s) must remain in the BOCT when this key is used (e.g. if no aura that the BOCT is driven through with the relevant key will accept the portion of the capture(s)), the remainder(s) is/ are added to the BOCT and given (a) new capture ID(s) as if that capture (e.g. portion of the sensory input/ output stream) were made on its own.

Register: (BOCT Driving) Public Show 📋

Register name: Public Show; Data type: Integer; Possible values: NULL, ‘1’, ‘2’

- The *BOCT driving* Public Show register is used to keep track of which of the two *BOCT driving* public show keys needs to be used next.
- When the BOCT is prime meaning that it contains no captures after having had done so, the value for this register is NULL.
- When the BOCT is not prime and ‘public show one’ has been used most recently, this register gets set to ‘2’ (two) meaning that ‘public show two’ needs to be used next.
- When the BOCT is not prime and ‘public show two’ has been used most recently, this register gets set to ‘1’ (one) meaning that ‘public show one’ needs to be used next.

Register: (BOCT driving) Public Hide 📋

Register name: Public Hide; Data type: Integer; Possible values: NULL, ‘1’, ‘2’

- The *BOCT driving* Public Hide register is used to keep track of which of the two *BOCT driving* public hide keys needs to be used next.
- When the BOCT is prime meaning that it contains no captures after having had done so, the value for this register is null.
- When the BOCT is not prime and ‘public hide one’ has been used most recently, this register gets set to ‘2’ (two) meaning that ‘public hide two’ needs to be used next.
- When the BOCT is not prime and ‘public hide two’ has been used most recently, this register gets set to ‘1’ (one) meaning that ‘public hide one’ needs to be used next.

Register: Polarity Override 📋

Register name: Polarity Override; Data type: Boolean; Possible values: FALSE, TRUE

- The Polarity Override register is used to induce the technique of overriding the polarity constraint for keys that operate on a polarity basis.
- The 'polarity override' key sets this register to TRUE.
- If and only if the register is set to TRUE, then the polarity constraint can be overridden. If the user sets the register to TRUE and utilizes any polarity based key in a way that does or does not override the default polarity constraint thereof, this register gets set back to FALSE.

Register: BOCT Priming 📋

Register name: BOCT Priming; Data type: Boolean; Possible values: FALSE, TRUE

- The BOCT Priming register is used to declare the BOCT as either prime or not prime.
- When a capture or captures previously resided within the BOCT and then the BOCT becomes empty, this register gets set to TRUE.
- When the BOCT begins to contain one or more captures, it becomes not prime and this register gets set to FALSE.

Register: Capture Break 📋

Register name: Capture Break; Data type: Boolean; Possible values: FALSE, TRUE

- The Capture Break register is used to split captures based on aura acceptance maximization.
- The ‘capture break’ key sets this register to TRUE.
- If and only if the register is set to TRUE, then the style of channeling captures into auras assumes a maximization strategy based on the creation time of the auras involved and regarding the specific BOCT driving key types’ functions (as described on page 35). Any channeling induced by a single key will set this register back to false, even if the register does nothing.

Table: Aura Activation 

Field Name	Activated_Aura_ID	Activating_Interaction_ID
Data Type	Integer	Integer

- The Aura Activation table is used to keep track of which **aura interaction** was involved in the activation of a particular aura. Aura interactions are discusses more thoroughly on page X.

- Here are the descriptions of the fields in this table:

  - Activated_Aura_ID: References Aura_ID from Auras. The identifier for an aura that is activated.
  - Activating_Interaction_ID: References Interaction_ID of the Aura Interactions table (defined on page X) to keep track of which aura interaction was involved in the activation of the coinciding activated aura.

Table: Capture Setting 

Field Name	Capture_Or_Partial_ID	Coinciding_Activated_Aura_ID
Data Type	Decimal	Integer

- The Capture Setting table is used to keep track of where the set capture(s) (or partial capture(s) in the event that the 'capture break' key has been called) will respectively go if a single consecutive private *BOCT driving* key is called should the state of this table persist. That is, if this table has a specific state (e.g. one or more rows are populated) and one of the two private *BOCT driving* keys gets called (e.g. show or hide), then every capture (or partial) as listed in the Capture_Or_Partial_ID column gets immediately channeled into the aura denoted by the Activated_Aura_ID column within that same row.

- Here are the field descriptions:

  - Capture_Or_Partial_ID: References Capture_ID from the BOCT table and in cases where the capture break register is TRUE, appends a decimal specifier to any capture (i.e. to the ID within this table) that is eligible to be partially channeled (if applicable) by referencing the Partial Compatibility table (defined on page X).
  - Coinciding_Activated_Aura_ID: The aura that the capture (or partial capture) will be channeled into if a private *BOCT driving* key gets called while the state of this table persists.

Table: Partial Compatibility 

Field Name	Original_Capture_ID	Partially_Compatible_Aura_ID	Accepts_Up_To	Break_Specifier	Actualizing_Interaction_ID
Data Type	Integer	Integer	Date/Time	Integer	Integer

- The Partial Compatibility table is used to keep track of which captures are partially compatible with which auras.

- The table is a prospective matching mechanism regarding what could happen if the user calls the 'capture break' key when channeling captures into auras (if applicable).

- This table lists all of the options that could actualize given the dynamic state of the BOCT and set of captures therein if the user calls the 'capture break' key however if the user does not call the aforementioned key, none of these outcomes would occur.

- Here are the field descriptions for this table:

  - Original_Capture_ID: References Capture_ID from BOCT table. The identifier for a partially compatible capture.
  - Partially_Compatible_Aura_ID: References Aura_ID from Auras table. The aura that a given capture is partially compatible to be channeled into.
  - Accepts_Up_To: The creation time of the aura which indicates the latest point in time the aura can accept up to from a given capture (based on aura acceptance constraints). *Note that this field is not necessary for the final database scheme given that the value is already being stored elsewhere, but it will likely be helpful for testing purposes.
  - Break_Specifier: This field is used to denote portions of a capture based on how many prospective auras there are to accept a portion of it. For example, if every portion of a capture can be channeled by a single subsequent private *BOCT driving* key given the current set of auras, there are 'X' portions where 'X' represents the number of auras that the capture requires to be fully accepted via the divvying of its portions. If the entire capture cannot be accepted via the divvying of its portions given the currently available aura(s), the number of portions is 'X' + 1 where 'X' is the number of auras that will respectively accept a portion of the capture and the additional portion is what goes back into the BOCT with a new capture ID. The portion (specifier) values are incremented (increased

by one) starting from the number one in a sequential order based on the creation date of the accepting auras. That is, if a capture is broken into four parts, the parts are numbered one through four (1-4).

- Actualizing_Interaction_ID: If a portion of a capture gets channeled into an aura, the corresponding row in this table does not get deleted (as other possible partially compatible capture portions that become no longer feasible to channel do) but rather, this field in the relevant row references Interaction_ID from the Aura Interactions table so as to keep track of the trace of a particular partial capture's channeling.

Keyset: Aura Referencing 🔑

- The *aura referencing* keys are used to reference auras that have been previously created. The aura referencing keys are used with *BOCT driving* keys as well as the *aura movi*ng keys (defined on page X) and the less common 'aura sight capture' key.

- These keys are used to 'point' to an aura (only one aura can be referenced or pointed to at a time) based on when the order in which the aura was created.

- There are two ways the user can reference a given aura. The first way is by counting back from the most recently created aura of a particular color (where the most recently created aura is one back, the second most recently created aura is two back, and so on). The second way is to reference the auras using instance notation which is set relative to BOCT priming. Instance notation in this context categorizes the auras based on their color and then numbers them according to BOCT priming where the first aura of color 'X' to be created after the BOCT becomes prime is denoted as X_One, the second is X_Two, and the most recent aura of that color to be created after BOCT priming is X_i (in computer science, 'i' denotes the most recent instance of something).

- There are two aura-reference-specific number keys that are specifically used to form the numbers that relate to an aura's **number code** which is the system of numbering the auras as described above. The two number keys for this purpose are:

  - 🗝 Aura Reference Number One
  - 🗝 Aura Reference Number Two

- From here, there are two different applying keys that apply the number which is created by one of these keys or by combining them in various ways using arithmetic (as defined on page X) in one of two ways:

  - 🗝 Aura Counting Reference Apply
  - 🗝 Aura Instance Reference Apply

- The ‘aura counting reference apply’ key is used strictly with whole, positive numbers formed by one of the *aura reference numbers* alone or by combining one or both of them in various ways using arithmetic. That key counts back ‘X’ auras of a particular color (where ‘X’ is the number formed using the *aura reference* numbers) or by default, it counts back one key if not paired with another number.

- The ‘aura instance reference apply’ key is paired with either a single number (formed the same way as before) or it is paired with a number within an equation relating to ‘i’. In the former case, a single number ‘X’ (when applied using ‘aura instance reference apply’) would be used to count forward ‘X’ auras starting from the time at which the BOCT last became prime. In the latter case, the ‘aura instance reference apply’ key applies an equation involving the letter ‘i’ by itself or with a subtraction therefrom of a number formed with the *aura reference* numbers in such a way that the most recent aura of a particular color is treated as instance ‘i’, the second most recent instance of that color aura is deemed instance ‘i’-1, and so on.

- One more key is going to be introduced with this keyset because this is the only set of keys that relates to it directly. This key is used to specify which aura color type is going to be paired with a specific reference applying key.

     Aura Creation Block: Blocks the creation of an aura and sets the related channel register to the color of the aura that it is paired with.

Aura Referencing Constraints 

- The *aura referencing* keys are relatively straightforward. The main consideration in terms of their function is what key or keys precede the aura reference.

- As stated previously, there are only three types of keys that pair with *aura referencing* keys. The three key types form a total of six distinct use cases.

- When only one of those particular types of keys is paired with aura referencing, the referencing process is very simple and operates normally. Below is the exhaustive list of possibilities reasons for referencing an aura.

    - Channel (a) capture(s) publicly into the light or dark of an aura
    - Derive (a) capture(s) from the light or dark of a non-moved aura
    - Move an aura to someone
    - Move an aura from someone to the self
    - Move an aura back to its default location from someone or the self
    - Capture the artificial sight of an aura.

- If two or more of the use cases listed above (excluding the last case which is defined below) are triggered prior to referencing an aura, the behavior only applies the first triggered function out of the non-disabled ones that remain when the reference is made and any other use case that has been triggered gets reset. This is also the rule that is applied when multiple relevant keys within a specific type alone (e.g. public *BOCT driving* keys) are called which would otherwise ambiguate the function of an aura reference. For example, if the user called ‘public hide one’ then called ‘public show one’ then referenced an aura, ‘public hide one’ would apply and ‘public show one’ would get deleted permanently. The user can disable keys to cause ones that wouldn’t normally apply be applied. For example, if ‘public show one’ was called then an aura moving key was called and then the user disabled the ‘public show one’ key then proceeded to reference an aura, in this case the aura moving key would be applied by the reference. There are two specific combinations of the use case keys listed above that are able to be applied with a single aura reference:

    - Move the aura back (i.e. to a non-moved position) and channel into it
    - Derive from the aura and move it (i.e. to or from someone)

❖ That is, if both use cases within one of the accepted combinations are triggered, both of those functions can be applied at once with a single aura reference and they will be applied in that order. If another use case is triggered additionally to either pair within these two accepted combinations, the third use case prevents the combinatory function and defaults back to only applying the first triggered from the remaining use cases.

❖ When an *aura reference applying* key is called, the following constraints apply:

→ Only the 'counting reference apply' key has a default application (one back).

→ A valid number created using the *aura reference number* keys pertaining to the 'aura reference counting apply' key is a non-negative, non-zero number.

→ A valid number created using the *aura reference number* keys pertaining to the 'aura reference instance apply' key is generally a non-negative, non-zero number. If the number 'X' is a part of an equation involving 'i' that takes the form 'i'-X, the number can be zero but it cannot be negative.

→ If more than one valid number is formed using the *aura reference number* keys and then an *aura reference applying* key is called, only the last of the numbers formed is applied. Any previously created number is deleted and never used.

→ If an *aura reference applying* key is called and for whatever reason, no aura gets referenced or if the reference does not apply any of the use cases, any *aura reference number* keys that have been formed or use case-triggering keys that have been called get deleted.

→ If a particular use case is triggered prior to there being a possible way of applying it (e.g. a public BOCT driving key called prior to the creation of any single aura) and thus it is not applied or if one or more use cases are triggered and then disabled and not re-enabled prior to an aura being successfully referenced and if the reference applies any other use case therefor, then the key(s) involved in triggering the use case(s) get(s) deleted and never used. If however, one or more use case-triggering keys are called then disabled and if a subsequent aura reference is made but none of the use cases are applied and then one or more of the use case-triggering keys gets re-enabled, all of said keys are still eligible to be applied using a subsequent aura reference based on the first-called-and-enabled rule described above.

→ To apply the ‘artificial aura sight capture’ key, said key must be called and then the aura that the user wishes to capture the artificial sight of must be referenced and no other referencing use case can be triggered (or would otherwise be applied) when the aura reference is made. If this key is triggered and a subsequent aura applies any other use case thereof, the non-applied ‘artificial aura sight capture’ key gets deleted and never used.

Register: Aura Color Channel 📋

Register name: Aura Color Channel; Data type: String; Possible values: *the eight aura colors

- The Aura Color Channel register is used to keep track of which type of aura (in terms of color) is going to be referenced when an *aura reference applying* key is called.
- When referencing an aura, this register has to be set to a specific color to reference an aura of that color.
- This register gets set automatically to the color of the last (most previous) aura type created.
- The user can set this register manually without creating a new aura by using the 'aura creation block' key followed by an *aura creating* key to block the creation of an aura while still setting this register to the color of the relevant *aura creating* key being (blocked and) called.

Table: Numbers 

Field Name	Number_ID	Instantiating_Key_ID	Value_ID	Applied_With_Key_ID
Data Type	Integer	Integer	Date/Time	Integer

- The 'Numbers' table is used to keep track of numbers that have been instantiated (created) using any of the 12 various types of numbers in the WaveLight framework.

- A given uniquely identified number is stored with only one row in this table and assumes a single value however, it is very common for numbers within one type (e.g. out of the 12) to be paired together to form a compound value.

- The process of forming a compound value using two or more number keys of the same type involves the use of *arithmetic-related* keys (defined on page X). Even when numbers are paired together this way, the 'number' itself has a single identifier ID (which is given to it upon detection of the first new number of a specific type that may or may not be paired with (any) other number(s)) and the pairing of the number keys simply changes the uniquely identified number's value.

- Here are the descriptions of the fields in this table:

  - Number_ID: The unique identifier for a given number.
  - Instantiating_Key_ID: The (number) key that instantiated the creation of the number which may or may not later be altered in value.
  - Value_ID: If applicable, references Value_ID of 'Values' table. If a number's value is altered (with other number keys of the same type), any alterations get stored in a different table wherein all relevant altering (number) keys have the same Value_ID.
  - Applied_With_Key: When a particular number gets applied using one of the corresponding keys that is able to apply it with regard to that specific number's type classification, this field in this table references Called_Key_ID from the 'Called Keys' table so as to specify that the number has been applied and (if applicable) what key applied it.

- Note: Any time that two or more numbers of the identical value are created either through calling a single number key or through the use of arithmetic (e.g. to form the

value three (‘3’)) and then an applying key that would apply at least one of them is called, only one gets applied and the other ones get deleted.

Keyset: Arithmetic operators 🔑

- Arithmetic operators are used with number keys to form different values than the number keys form on their own.

- It is uncommon to need to form a value that exceeds roughly five units less than or five units greater than the respective minimum/ maximum number keys regarding a given type of number. With that being said, the need to create compound numbers in general is very common. For that reason, addition and subtraction operators are included in the WaveLight dictionary of keys, whereas multiplication and division currently are not.

  - 🗝 Arithmetic plus: Adds the number after to the number or value before.
  - 🗝 Arithmetic minus: Subtracts the number after from the number or value before.

- With the addition of the arithmetic operator keys arises a use case that creates the need for a number zero (‘0’) key. Next, we will introduce the number zero key that is universal to all WaveLight numbers meaning that it can be paired (e.g. using arithmetic) with any other type of key. It can also be called on its own in order to step out of equations.

  - 🗝 Counting number zero (‘0’)

Arithmetic Constraints 📋

- It is possible to have two or more different arithmetic equations started without any previous equation having had completed. Finalizing the value produced by an arithmetic equation is achieved by applying the value up to that point (by using a key that is eligible to apply a number of a particular type) or by 'stepping out' of the equation and then beginning a new equation using arithmetic operations on the same *number* key type.

- Stepping out of an arithmetic equation is a technique used to pause the value-creation process that building an equation ultimately achieves and this is done by either calling a number key with a different type of number classification or by calling a number of the same type without linking it (e.g. automatically) with the ongoing equation.

- Let's imagine that keys 'A' and 'B' are counting number keys and that keys 'C' and 'D' are instance number keys.

- To step out of an arithmetic equation (e.g. 'A', '+', 'B',…) by calling a number key with a different type of number classification (e.g. …'C',…) allows the value being formed with the equation to remain as is without being finalized. If the number with a classification different from the numbers within the equation is proceeded by a subsequent arithmetic operator (e.g. …'+',…), this situation is ambiguous regarding whether a new equation will begin with the most recently called number (type), or whether the equation from before (with the counting numbers) will be altered with another number of the same type. For example, to proceed with …'D',… creates an equation of instance numbers whereas continuing instead with …'A',… (for example) picks back up with the previous equation involving counting numbers. Also, even if the user begins a new equation involving a different type of *number* key (as in the case of proceeding with key 'D'), the user can 'step back into' the previous equation (in this example, the counting number equation) by using an arithmetic operator followed by a *number* key of the relevant type.

- It is possible that an arithmetic operator may never get used in cases where there are no *number* keys for it to function with (which may lead to it getting automatically deleted) or when an arithmetic operator has been disabled. Arithmetic keys (plus and minus) are only paired with a specific number or value after the condition that

an unapplied number (or a set thereof that was used to form an unapplied value) precedes the operator and a number of the same type immediately proceeds the operator prior to any other type of number being paired using that specifically called operator key first. An arithmetic operator that is not exposed to this exact criterion remains unpaired. Further, when an arithmetic operator is detected, it can only ever be used to pair (preceding) *number* keys that are enabled at that moment with any proceeding number keys that; determined when the (respective) detection(s) thereof occur(s), have been called while the arithmetic operator is enabled. The moment that an arithmetic operator is deemed unusable due to any of the constraints disqualifying its use cases, it gets permanently deleted.

- To step out of an arithmetic equation by calling a number key with the same type of number classification behaves similarly to the other 'step out' method. If an equation with a specific number type is ongoing (e.g. 'A', '+', 'B',…) and then a subsequent number of the same type is called and that number is not automatically linked with the ongoing equation (e.g. …'A',… (without a preceding arithmetic operator)), the work within the equation is paused in the sense that the only way to step back into the equation is by disabling any proceeding number keys with the same number type as the numbers within the equation without any arithmetic operators being used to pair the subsequent number keys together. This constraint stands true no matter how many subsequent number keys of a particular type are called that are not a part of the previous equation involving the same *number* key type. For example, if the sequence above continued as …'B',… and then after that, the user disabled the two most recent keys (e.g. the 'A' and 'B') that were called after the equation was stepped out of then proceeded to call an arithmetic operator followed by another number key of the same type, the original equation could continue, picking up where it left off up to the point of being exited. However, if one of the two arithmetic operators is used to pair any two subsequent number keys that are of the same type as the *number* keys within a previous equation that has been exited but where the value thereof has not been finalized, the exited equation is final and cannot be altered any further.

- There can be as many non-finalized equations as there are types of numbers – one for each type. Simply put, to start a new equation using arithmetic on two or more numbers of a specific type finalizes all previous (e.g. separate) equations involving numbers of that same type.

- While there can be multiple incomplete arithmetic equations occurring at once, there cannot be two eligible arithmetic keys at one given time (think: a key that has started to perform its function but has not completed it). This is due to the fact that

the type of preceding number or value that a given arithmetic key will be linked to (or whether it will be applied at all) is not determined until both numbers (or the value and the number) that it will be applied to are both detected (as was shown in the example sequence: ‘A’, ‘+’, ‘B’, ‘C’, ‘+’, ‘D’/’A’).

- Calling one type of arithmetic key after the other type has been called when the former operator is possibly going to be paired with a number or value that preceded it and has also not completed its function, (e.g. if an unapplied number or value exists and prior to its application (e.g. with a *capturing* key), ‘arithmetic plus’ is called and then immediately thereafter, ‘arithmetic minus’ is called, the effect of the most recently called operator key on the preceding is that it) effectively disables all non-applied arithmetic keys that precede it. Only the last operator key called (prior to the relevant subsequent *number* key called which solidifies the operator’s function) is used to pair numbers. However, if (a) proceeding arithmetic operator(s) get(s) disabled, the preceding ones that have not been deleted already automatically get re-enabled as long as none of the operators have been applied. For example, ‘A’, ‘+’, ‘B’, ‘+’, ‘-‘,… followed by disabling the ‘+’ (‘arithmetic plus’ key) followed by …’A’,… results in the equation ‘A’ + ‘B’ – ‘A’. When two or more arithmetic operators are called prior to the application of any one operator’s function, upon application of the called, eligible operator key, all non-applied operator keys get deleted.

- A unique constraint appears when a situation like ‘A’, ‘+’, ‘B’, ‘+’, ‘C’,… (where an equation is stepped out of using a number key of a particular type proceeding an eligible arithmetic operator being called within a non-finalized equation involving two or more number keys of a different type) because there exists a particularity regarding the ‘arithmetic plus’ operator which one might assume would be discarded according to this sequence of keys. Instead, the constraint is such that the arithmetic plus operator can still be applied in a situation where no subsequent equation (involving any *number* key type) is created. The intuition here is that the situation is not ambiguous in the sense that the plus operator is neither applied nor discarded according to the current constraints. So if the user proceeds the sequence with a counting number key (e.g. …‘A’), then the addition operator is applied such that it adds the most recently called instance of ‘A’ to the ongoing equation to form ‘A’ + ‘B’ + ‘A’. Again, ‘hanging arithmetic operators’ like observed in this example are only salvaged in situations where no subsequent, new equations involving any number key type are started. Namely, if a subsequent arithmetic operator has not been applied. Also, if the user calls either of the arithmetic operators at any time following the detection of the arithmetic operator that would otherwise remain eligible through the ‘hanging arithmetic operator’ constraint, the ‘hanging’ operator is

discarded in place of subsequent operator key calls, even if the subsequent arithmetic operator(s) get disabled prior to be applied.

Table: Values 

Field name	Value_ID	Key_ID	Number_Type
Data Type	Integer	Integer	String

- The 'Values' table is used to keep track of any ongoing or completed formation of compound numbers formed using (an) arithmetic operator(s) combined with number keys of a specific type.

- Every *number* key that is linked with another of the same type via *arithmetic* keys will share the same *Value_ID* with the other *number* key(s) as well as the arithmetic key(s) involved in the equation it is a part of.

- This table serves to piece together the components that form the equations that are used to create values other than those which the respective number key types have keys for by default.

- The fields in this table are:

  - Value_ID: This is the identifier for a given component (e.g. a *number* key or an *arithmetic* key) involved in a particular equation.
  - Key_ID: References Called_Key_ID from 'Called Keys' to identify which called key is involved in a particular equation (e.g. assuming a particular Value_ID).
  - Number_Type: The type of *number* key that that a particular component of an equation is categorized as or is paired with regarding *arithmetic* keys.

Register: Arithmetic Operation 💾

Register name: Arithmetic Operation; Data type: String; Possible Values: Null, Plus, Minus

- The Arithmetic Operation register is used to keep track of which arithmetic operation (if either) will be applied to a subsequent number that is produced of an eligible type.

- This register works hand-in-hand with the Arithmetic Eligibility register to determine how the equation building process should unfold. The Arithmetic Eligibility key keeps track of the types of non-applied number(s) (or non-finalized value(s) resulting from an ongoing equation) that are eligible to be paired using arithmetic and if the Arithmetic Operation register is not set to Null, an operation will be performed on the next *number* key of an eligible type that gets detected.

- The Arithmetic Operation register gets set according to the constraints defined in the last section.

Register: Arithmetic Eligibility 💾

Register name: Arithmetic Eligibility; Data type: Integer; Possible Values: 0-2,047

- The Arithmetic Eligibility register is used to keep track of which number type has existing numbers or values that are eligible to be paired with arithmetic operators to form a new equation (e.g. if an arithmetic operator key has been called and then a subsequent number key of the same type gets called).

- This register stores an integer (whole number) value that specifies which number type(s) if any (of the 12) are eligible in that sense.

- The integer value that this register will be set to is determined based on the following logic:

  - If none of the 11 number key types are eligible, the register is set to '0'.

  - If the first of the 11 number type sets (regarding the order the sets are introduced in this manual) is eligible and only that number type is, the register gets set to '1'.

  - If the first and second of the 11 number type sets are eligible and only those two number types are, the register gets set to '2'.

  - This pattern repeats until the last combination of the types is formed which denotes that all of the number types are eligible in this context, in which case the register gets set to '2,047'.

  - The values this register can assume are preset to specific combinations of the number types such that (once more) the first number type and the second number type (and only those two) being eligible is preset to the value '3', the first number type and the third number type (and only those two) being eligible is preset to the value '4', the first number type and the fourth number type (and only those two) being eligible is preset to the value '5'.

- After the first number type has been individually paired with each of the other number types and each of the other respective number types has individually been paired with each of the other number types on its own as well (e.g. type two with type three, type two with type four, and so on), the next preset register value is determined for the case where the first, second, and third number types (an only those three) are eligible. From there, the next preset register value accounts for the first, second, and fourth number types being eligible.

- After every combination of three number types has been covered and assigned a value using this combinatory ordering, the sets of four, five, and so on all the way up to 12 are formed and given a preset number.

❖ Intuitively, the possible values this register can assume is simply determined by a statistics formula that calculates the number of combinations of eligibility regarding the 11 types of number keys. The purpose of describing the possible values of this register in this much detail is to disambiguate the matching of the possible register values to the specific combination of eligibility (given that it could be arbitrary).
❖ This register is automatically assigned based on how the *number*/ *arithmetic* keys are called and how the key-calling affects arithmetic eligibility for the respective number types.

Numbers and Values Constraints 

❖ Mostly everything regarding numbers has been covered, however there are some particular use cases involving numbers that require further disambiguation. Here are the technical constraints regarding the application of numbers and values:

→ Non-applicable values are ones that the user has formed using arithmetic which always necessarily yield a null reference regarding any possible application thereof. For example, if the user performs arithmetic on *BOCT marking* numbers to form the value '-1', that value is always necessarily non-applicable to the *BOCT marking* keys due to the functionality thereof. Any always necessarily non-applicable values which the user creates and then finalizes (by creating a separate, subsequent equation involving the same number type) gets automatically deleted. Below is a list of the types of number keys that have predefined non-applicable values along with the values themselves. Many of these number types have not been described yet, so do not worry about knowing what they are used for. The following list is mostly just for referencing later on:

- Aura referencing numbers: Negative values, zero, and any value (at a given moment) that exceeds the number that represents the highest aura code value of a particular aura color that is non-containing.
- Key component numbers: Negative values, zero.
- BOCT-marking numbers: Negative values, zero, and any value (at a given moment) that exceeds the number of captures within the BOCT.
- BOCT capture numbers: Negative values, zero, and any value (at a given moment) that exceeds the number of captures within the BOCT.
- Tower Refract looping numbers: Negative values, zero, and any value (at a given moment) that exceeds the number of available Tower Refract loops.
- Aura-deriving numbers: Negative values, zero, and any value (at a given moment) that exceeds the maximum number of captures within any one non-closed, containing aura.
- Flag-pointing numbers: Negative values, zero, and any value (at a given moment) that exceeds the number of flags within any one non-closed, containing aura.

→ The other three types of numbers do not present the opportunity to create always necessarily non-applicable values. With that being said, not all values that

are unapplied upon the detection of a key that could otherwise apply them are necessarily non-applicable and might just be non-appliable at a given moment.

→ Non-appliable values are those which have been created that are not non-applicable but for whatever reason, are not applied by a key that could otherwise apply that number (e.g. given the type thereof) upon detection of the applying key. For example, if the user forms a value using the *key component* number keys and then tries to apply it with the 'enable key component' key but that key is already enabled, the number is non-appliable in that sense. Below is a non-exhaustive list of the number key types that may involve momentary non-appliable values and examples of how the values could be non-appliable.

- Key component number keys: Using a key component applying key with a number in such a way that the applying key does not get applied because the function of the applying key contradicts the key component state of the key the user is attempting to apply it to.
- Word search number: Attempting to apply a positive number in the word search process where such word has not occurred following the most recent priming of the BOCT.

→ When a non-appliable value that is not necessarily always non-applicable is unused two times by any key that could possibly apply that same value in different circumstances, the value gets automatically deleted. The first time such key is detected when such a value has been created and the key does not apply the value, the default (e.g. one back) function of the key (if it has one) is blocked. The second time, the default behavior does not get blocked. For example, if three keys were called (e.g. 'A', 'B', and 'C'), then 'key component disable' was called (disabling key 'C' by leveraging its 'one back' default function) then the value '3' was created using key *component number* keys then the 'key component enable' key was called (this sequence would attempt to enable key 'B' (according to the function of key component number keys as described on page X) however), because key 'B' would already be enabled, the default function of the 'key component enable' key would be blocked by the non-appliable value '3'. If from there, the user leveraged the 'key component enable' key again by calling the 'duplicate' key, the '3' would be attempt to apply 'key component enable' to 'C' which it would do successfully as that key was previously disabled (it is recommended to study the *key component* keys section to understand how the trace ensues within this example). However, if after the first call of 'key component enable', the user proceeded to call 'key component disable', that would be the second time the '3' would be considered but non-applied (because

'C' is already disabled), so the value would get deleted. In this scenario, the key component of the first 'key component enable' key called in this sequence would be targeted by the 'key component disable' key's default (one back) behavior (because it did not get blocked) however, the function of that 'key component enable' key would be complete and thus, nothing would happen.

→ If a particular key is involved in considering a particular non-appliable number or value (e.g. causing that non-appliable number or value to be considered for the first time), and then that applying key gets disabled, if there are any new numbers or values called or created, if the already considered number or value does not get disabled, and if the disabled applying key gets re-enabled and for a second time does not successfully apply any of the numbers, then the applying key does not have its default behavior blocked. The default behavior applies, and the non-appliable numbers get deleted. If instead, no new numbers or values were created then even if the already once considered number or value had not been disabled, the applying key could be disabled and re-enabled many times and its default behavior would still be blocked upon re-enabling it.

→ If an applying key is called but it does not apply any number(s) or value(s) and its default behavior is blocked because of this and then the applying key gets disabled, if there were disabled numbers or values that were called and not applied prior to the applying key being called that get re-enabled or if a number or value that were possibly disabled or were just non-appliable at the time of the applying key's detection get altered in such a way that the number or value becomes appliable (including the already once considered one) and then after that, the applying key gets re-enabled, then the applying key can apply the number(s) or value(s).

→ This constraint applies only when the value has not been altered since the first non-appliable consideration and also, if any additional non-appliable value(s) of the same number key type is/ are created and then a key that could possibly apply a group of non-appliable values but in this situation, cannot apply any one of them; is called, the existence of the first non-appliable value that has already been considered but not applied causes not only the default behavior blocking mechanism regarding the non-appliable values to be removed (e.g. the default behavior is not blocked) and also, both (or any amount of subsequent) non-appliable values (of the same type) get deleted. Finally, any non-appliable number that is present when a key that could apply it is called and the key that could apply it applies a different number but not it through a process that does not directly leverage the applying key's default behavior (even if the number it

applies matches the value used within the default behavior), any non-applied value or number is deleted and never used.

❖ The process of disabling or manually deleting a value allows the user to prevent that number key from being used by a relevant applying key. Here are the constraints regarding disabling or manually deleting numbers or values.

→ If a number key is called and it has not been paired with any arithmetic operator to form a different value, disabling the number using the *key component* keys is allowed and the number can be re-enabled and applied later on.

→ If an equation is ongoing and no subsequent equations of any number type or individual numbers of a different type have been formed or called respectively, disabling any one *number* key involved in that equation removes the number in a complementary way to how it was introduced- by subtracting the number from the equation's value if the number was originally added or by adding the number to the equation's value if the number was originally subtracted. If the user steps out of the equation, this technique can no longer be applied to that equation, even if the equation's value is not finalized and even if all subsequent number keys called (regardless of the type) get disabled (i.e. even if the user could technically step back into the equation).

→ If an equation is being used to form a value and the equation has been stepped out of then regardless of whether the value has been finalized or not or even if the equation is stepped back into and altered, if the user disables any one key involved in the equation then all involved keys (i.e. the arithmetic operator(s) and the *number* keys) get disabled. If the user wishes to re-enable those keys, they must re-enable the same key that they disabled before.

→ If a number key is called or an equation is used to form a value then a subsequent equation involving the same number type (which through creating, finalizes the value of any preceding equation of the same type) can be used to 'land on' a number or a value that is the byproduct of a previous equation using an arithmetic operator paired with exactly two number keys of that type and from there, the user may 'drive to zero' and then step out of the latter equation to permanently delete both numbers. For example, imagine the user had called '1', '+', '2' (using *counting number* keys, for example) to form the value '3' or even that they simply called 'counting number three'. The user could land on the value '3' with any two number keys of that type paired with either arithmetic operator (e.g. '5', '-', '2' or '1', '+', '2' or '3', '+', '0') and then proceed to apply one or more subsequent arithmetic operations as a continuation of the second equation to

eventually form the value zero and then from there, to step out of the latter equation permanently deletes both values resulting from both equations.

Table: Aura Interactions 

Field Name	Interaction_ID	Involved_Aura_ID	Use_Case_Key_ID	Interaction_Type
Data Type	Integer	Integer	Integer	String

- The 'Aura Interactions' table is used to keep track of the different interactions that occur with auras that the user has created.

- The table keeps track of references to auras that involve any of the use cases listed in the *Aura Referencing Constraints* section on page X (excluding the use case involved in capturing the artificial sight of an aura).

- That is, any time that an aura reference actually involves (an) interaction(s) with (an) aura(s) (sometimes references do not involve any interaction (such as in cases when there are no triggered use cases upon the reference being made) and it is also possible that a single aura reference will involve two interactions with the aura being referenced), the interaction(s) are stored in this table.

- Here are the descriptions of the fields in this table:

  - Interaction_ID: The identifier (ID) for a particular interaction is stored in this column. At most two rows in this table will share a unique Interaction_ID (because at most two interactions can occur with a single aura reference).
  - Involved_Aura_ID: References Aura_ID from the 'Auras' table. The ID of the aura involved in the interaction.
  - Use_Case_Key_ID: References Called_Key_ID from 'Called Keys' table. The ID of the key that triggered the use case involved in the interaction.
    - Interaction_Type: The description of the interaction. Possible values for this field are: 'Accepted into light', 'Accepted into dark', 'Derived from light', 'Derived from dark', 'Moved to *user_name*', 'Moved from *user_name*', and 'Moved back to default location'.

Table: Capture Movement 

Field Name	Moved_Or_Partial_Capture_ID	Moving_Interaction_ID	Finalization
Data Type	Integer	Integer	String

- The ‘Capture Movement’ table is used to keep track of the movement of captures or portions thereof to and from the BOCT.

- Any time an interaction involves the movement of the entirety or the portion of a capture, the bind between the moved (portion of the) capture and the interaction is tracked using this table. The table essentially serves to determine where a capture is at a given moment.

- Here are the descriptions of the fields in this table:

  - Moved_Capture_ID: References Capture_ID from the ‘BOCT’ table. The capture that has been moved.
  - Moving_Interaction_ID: The interaction that moved the capture.
  - Finalization: Whether the movement of the capture which has been moved according to the way that the interaction denoted by Moving_Interaction_ID caused it to (i.e. into what location and (if applicable), what type of movement) is final (at a given moment) or not.

Keyset: Counting Numbers 🔑

- Counting number keys are the vastest set of number keys as they are used with three different types of keys: (standalone) *capturing* keys, *word-search* keys, the 'thought of a person capture' key, *reference point* keys, and *step* keys (most number key sets only operate with one applying-key type, and there is only one other number key type that operates with more than a single applying-key type).

- There are five counting number keys:

  - 🗝 Counting number zero ('0')
  - 🗝 Counting number one ('1')
  - 🗝 Counting number two ('2')
  - 🗝 Counting number three ('3')
  - 🗝 Counting number four ('4')
  - 🗝 Counting number five ('5')

- Counting numbers are paired with *capturing* keys in order to 'count back' in distinguishable events in order to capture them. Counting numbers precede the capturing key to apply the number key(s) that have been called to make the capture. For example, the 'sound in real life from self' capturing key defaults to capturing the last (i.e. most recent) word back. If instead you called 'counting number five' and then immediately proceeded to call 'sound in real life from self capture', this sequence would capture the fifth word back (out of those spoken by the self). Counting numbers are able to be paired with eight out of the 13 total capturing keys. Only the standalone capturing keys (as defined on page X) can be paired with the counting numbers because those are the only capturing keys that target channels wherein distinguishable events occur.

- The counting numbers are designed to be compiled together in such a way that when applied with *capturing* keys, multiple captures can occur at once. For example, if you call two different counting number keys and then proceed to call a capturing key, then the two counting number keys will both be applied, and two separate

captures within the channel specified by the capturing key will be made- one for each value represented by the counting numbers. As soon as a capturing key is called, all (appliable) counting numbers that have not already been applied to a capture will be applied to that specific capturing key.

- The three sound channel capturing keys are noteworthy in this context in the sense that they sometimes get tricky when paired with counting numbers. The counting numbers strictly target real words when applied to sound capturing keys (e.g. words that are defined in a standard dictionary). For example, if you spoke (out loud in real life) the sentence "I'm doin' well today" (i.e. 'doin' rather than 'doing') and then proceeded to call the following three keys (in this order): 'counting number three', 'counting number two', 'sound in real life from self capture', then the words "I'm" and 'well' would be captured, rather than the words "doin'" and 'well'.

- *Counting number* keys also pair with the *word-search* technique (defined on page X) which is used to filter the word which sound-based capture keys target when called. For example, if the word search register (defined on page X) got set to the word "I'm" and then the user called 'counting number one' and then proceeded to call 'sound in real life capture' (assuming the sentence from above was spoken aloud), the *capture* key would count back one instance of that specific word rather than any word.

- This number type can also be used to count back regarding reference points (as defined on page X) that are created manually by the user. The process of targeting reference points with counting numbers is straightforward and simply involves creating the number that corresponds to the number of reference points back at any given moment. The only consideration to keep in mind when going through this process is that only one *counting number* key (or value produced with arithmetic thereon) can be applied by the 'reference point seek' key meaning that if there is more than one counting number created, only one will apply. In this situation, the first one that was called is the one that will get applied and the other keys which are thus considered but non-applied follow the two-considerations-with-zero-applications-to-get-deleted constraint as described in a previous section.

- To reference a thought of a person using counting numbers involves thinking of that person again (which sets the person thought register to that person) and then applying a counting number or multiple numbers using the 'thought of a person capture' key which then counts back the number that the value or values specify regarding instances of that specific person being thought of.

- Finally, counting number keys are used to specify the number of units that a *step* key should step. The *step* keys (defined on page X) are used to step forward or backward (some number of units) regarding a particular type of construct in the WaveLight framework whether that is a capture, an aura reference, a key component alteration, or something else.

Keyset: Instance Numbers 🔑

- Instance number keys are the other set of number keys that pair with more than one kind of applying-key type, namely: the *word-search* keys, 'thought of person capture' key, and the 'reference point seek' key.

- There are two instance number keys:

  🗝 Instance number one ('1')

  🗝 Instance number two ('2')

- Instance number keys are paired with *word-search* keys to make sound-based captures targeting particular words. Instance number keys are different than counting number keys in that they target 'instances' of distinguishable events in a forward-moving sense, rather than backward. Instance numbers are easier to understand when considered using a specific example.

- If the word 'world' was spoken three times (e.g. all from within the same sensory channel), then the instance number keys (assuming the word for search register was set to that word) could be used to specify which of those instances of the word you want to capture. The way the instances of words spoken are numbered is based on the activity within the BOCT table. As soon as something has entered the BOCT table after a period of time wherein it was completely empty (i.e. when the BOCT is prime), instances of specific words being spoken begin being denoted at the number one (i.e. the first time that a word is spoken after the BOCT becomes prime, that word is deemed the first instance thereof). The logic behind instance numbers is a little more complex than each word being assigned a number, but in practice it is very simple to understand. Let's dive deeper into this specific example.

- Let's say the sentence "Hello world." was spoken aloud while the BOCT was empty and then the word 'Hello' was captured using 'counting number 2', 'sound in real life capture', and then the capture was channeled into an aura (thus making the BOCT prime) and then immediately after that, the sentence "How is the world today?" was spoken. Take note about how the first instance of the word 'world' was spoken before the BOCT became prime for the most recent time, and how the second instance was spoken after the BOCT became prime.

- To trivialize things, let's introduce the idea of denoting the instances of the word 'world' using the letter 'i' (commonly used in computer terms to denote the 'most recent instance' of something). Intuitively, the second most recent instance would be denoted as instance 'i-1', the third most recent as 'i-2', and so on. More often than not, words will be captured in WaveLight using the word-search technique targets instance 'i' thereof which is the default applying behavior thereof (when no relevant numbers are called). If it helps to think in terms of assigning each instance of a particular word a number, that is another way to target specific instances of particular words. Every time that the BOCT becomes prime, the next instance of a particular word is the first instance and at any given time, the most recent instance of the word is instance 'i' (the first instance can simultaneously be the i'th). Any instance of the word that occurred prior to the most recent priming of the BOCT becomes assigned a negative value. The priming process is essentially just how the '0 point' is defined (although the number '0' (or zeroth instance) of something is never used to specify captures in WaveLight) and all words' numeric instance value(s) are relative to that point moving forward. In our example with the two instances of the word 'world', the former would be denoted as instance '-1' and 'i-1' and the latter as '1' or 'i'. Keep in mind that the '-1' in this case is not related to the '-1' in 'i-1'. The '-1' by itself is relative to the 'zero point' whereas the '-1' in 'i-1' is relative to 'i'. Only in this specific case where there exists an instance of a particular word immediately preceding and immediately following the priming of the BOCT does this partial ambiguity exist. Essentially, the denotation involving 'i' is used how the counting number keys are- to count back from the most recent occurrence whereas the zero point (determined with respect to BOCT priming) pairs with instance numbers to frame instances of common events in a numeric sense.

- Instance number keys are the only number key type that can be paired with specialty 'instancing keys' (as defined on page X which includes a key that denotes instance 'i') in order to apply arithmetic relative to instance 'i'. The specialty instance keys themselves can also be applied on their own because any notation involving 'i' (as they all do) is technically a dynamic number in the grand scheme of things. In fact, any time that a specialty involving 'i' is called and is applied on its own or if it is paired with instance numbers using arithmetic, the application thereof is applied as if it were an instance number. These keys even follow the non-appliable constraint defined previously.

- To circle back, the way that a user could target the former instance of the word 'world' from our example using instance numbers, the user could either form the value '-1' using arithmetic or they could use arithmetic (relative to instance 'i') to

create the necessary value (in this case, the user would subtract one from ‘i’). To capture the latter instance of the word ‘world’ within the example, the user could use ‘instance number one’ or the key that denotes instance ‘i’. Intuitively, if there were a third instance of the word ‘world’ being spoken that proceeded the aforementioned two, then that instance could be captured using the ‘instance number two’ key and once again the key that denotes instance ‘i’ would work too.

- To reference a thought of a person using instance numbers follows suite with the technique used to search for specific instances of a particular words. Again, the zero point at any moment is defined by the when the BOCT last became prime. The first thought of a specific person after the most recent priming of the BOCT is instance one, the second is instance two, and the numbering works the same way (but in the opposite direction) for instances of a thought of a specific person that precede the most recent priming of the BOCT. Using ‘i’ notation also works with this type of number application, where the most recent thought of a specific person is instance ‘i’ thereof.

- The process of applying instance numbers to reference point keys is very simple. It’d be analogous to there only being one word to ever needed to be searched for. When the user creates one or more reference points, they are denoted using the same numeric system that specific instances of particular words are denoted with. When the BOCT becomes prime, that sets the zero point. The reference points immediately before the most recent priming gets denoted as instance ‘-1’ as the reference point that immediately proceeds the prime state is denoted as instance ‘1’ and (intuitively) any subsequent or preceding reference point(s) are denoted accordingly using consecutive, sequential incrementation or decrementation (adding or subtracting one (‘1’)) respectively.

Keyset: Key Component Keys 🔑

- The *key component* keys are used to work with the key components of previously called keys. In particular, a key component of a given non-complete key (assuming eligibility) can be disabled, re-enabled, or duplicated.

- Intuitively, disabling a key component can only be done on the keys which have functions that do not get performed immediately upon detection, or which have otherwise not completed. *Number* keys, *arithmetic* keys, *BOCT driving* keys, etc. are a few examples of keys that do not immediately perform any of their core function upon detection (e.g. *number* keys require pairing with an applying key) and thus, can be disabled so long as their function has not completed. Some keys operate differently in that they may begin part of their core function upon detection but do not necessarily complete their function until some proceeding key(s) are called. Examples of these keys are 'Moment Link' and 'Tower Refract'. These keys operate differently when disabled, as is described in the sections discussing those respective keys. Then of course, there are situations such as when keys that would otherwise immediately complete their function upon detection have their function blocked (such as when a capturing key is called with non-appliable numbers present), and in those situations it is certainly possible for those keys to be disabled.

- Eligibility for a key to be re-enabled generally requires only that the key is already disabled, however the time at which a key is originally disabled and also when it gets re-enabled can affect the function thereof.

- There are two keys used to disable and re-enable key components respectively:

  - 🗝 Key Component Disable: Disables key components.
  - 🗝 Key Component Re-enable: Enables key components.

- The two keys listed above operate on a default (one-back) behavior as is common with other applying keys. The key component keys also pair with key component number keys. There are two:

  - 🗝 Key Component Number One ('1')

🗝 Key Component Number Two (‘2’)

- The key component number keys operate slightly different than the user may expect. Because these numbers keys are specifically used for counting back in keys called, using one of these keys, let alone several to form the right value can quickly make it confusing to find the right value to pair with one of the relevant applying keys. For this reason, key component number keys do not count as called keys (when considering how many called keys back a particular key is) until the number or value being formed is applied, or if a subsequent number key is called or if a subsequent value is formed after one number or value have already been.

- For example, let’s say that keys ‘A’, ‘B’, and ‘C’ have been called and that the user wants to disable the key components of keys ‘A’ and ‘B’. Here are two ways the user could do that. The first way would be to disable one key at a time (using the ‘key component disable’ key twice). The user could start by disabling key ‘A’ by using arithmetic on the key component number keys: ‘1’, ‘+’, ‘2’ which would not change the fact that key ‘A’ is the third considered called key back (even though three keys have been called after ‘C’). From there, the user could call ‘key component disable’ which would then apply the value three (‘3’) to the considered called keys back, thus disabling key ‘A’. After a key component number key or set thereof (used to form a value) have been applied (or gets finalized in the case of values), that number or value is considered as a single considered called key back. This process is known as ‘arithmetic compression’ and only ever occurs with equations involving key component number keys (including the arithmetic keys involved in the equation). To finish our first example, now that the user has performed the arithmetic and applied the value, there are technically two additional considered called keys back that: the value three (‘3’) and the ‘key component disable’ key. The makes it such that key ‘B’ is now four keys back, so the user could proceed to target that key with arithmetic on the key component number keys such as ‘2’, ‘+’, ‘2’ followed by ‘key component disable’.

- The second way the user could disable both ‘A’ and ‘B’ is by disabling them both at once (with a single call to ‘key component disable’). The user could start by creating the value ‘3’ using arithmetic on the key component number keys. When the user begins to form a new equation on the key component number keys (thus, finalizing the value of the first equation in this example), the equation as a whole that was used to create the value three (‘3’) is counted as a single considered called key back. As of this moment, ‘A’ is four back and ‘B’ is three back, rather than three and two respectively (as they were before). The user would need to continue to create the

value four (‘4’) and then by calling ‘key component disable’ with both values created, they can target both keys.

- In general, when targeting two or more consecutive keys, the user can effectively capture all keys with minimal confusion by using the tactic that involves aiming for the first key in the consecutive sequence of keys (regarding the number of considered called keys back that it is originally) and then immediately create the necessary subsequent values by aiming for that same key again and repeat once per the number of keys (so that there are the same number of values created as there are keys in that consecutive sequence). Of course, there are other strategies to achieve the same outcome.

- Unlike other numbers, key component values get finalized simply by stepping out of an equation of key component numbers. Due to the nature of their use case, equations involving key component numbers cannot be stepped back into When key component numbers are applied, they do not target only enabled keys. They target all considered called keys back which includes any finalized key component value or non-applied key component number which has subsequent key component numbers or values. For example, if the user calls ‘key component number two’ then proceeds to call ‘key component number one’, the latter key called technically targets the former (which can be disabled that way – although doing so and then proceeding with an arithmetic operator followed by additional key component number keys does not add onto the first equation). Key component number keys are very particular in this way. Even if the user calls a non-applied arithmetic operator such as in the case of a sequence (with key component numbers) such as ‘1’, ‘+’, ‘2’, ‘+’… and then applies the key component disable or re-enable key, the ‘hanging’ arithmetic operator counts as one key. This can be prevented by adding the universal ‘0’ key to the end of the equation. The key component number keys are tricky because any key that breaks a coherent equation thereof (e.g. that starts with a key component number and ends with one of the same type and only includes arithmetic operator keys in between) finalizes the equation and the subsequent key(s) as considered keys back. The only exception to this general equation-break rule is when called, unused arithmetic operators get automatically disabled from within (e.g. not at the beginning or end of) an equation involving key component number. For example, with our original equation of three keys: ‘A’, ‘B’, ‘C’, the following sequence of keys: ‘key component number one’, ‘arithmetic minus’, ‘arithmetic plus’, ‘key component number two’, ‘key component disable’ applies the value three (‘3’) to disable key ‘A’ however, as soon as that value three is applied, the equation separates any unused arithmetic operator(s) within the equation from the equation’s value which gets counted as a single considered called key back. That is, a situation like this would

now result in this sequence up to this point: ‘A’, ‘B’, ‘C’, ‘arithmetic minus’, ‘(compressed) key component value three’, ‘key component disable’. Any unused arithmetic operator keys are placed before the value of the equation in the order that they were called. This situation can be prevented by adding a ‘0’ between arithmetic operators in the middle of the equation or at the end.

- Finally, this set includes the ‘duplicate’ key which is used like copy-paste on a computer to simply repeat the key that immediately preceded it.

 Duplicate

---

Keyset: Alphabet Letter Keys 🔑

- The *alphabet letter* keys are used to set the word for search register so as to search for a particular word that was spoken.

- Alphabet letter keys are used to spell out the word letter-for-letter and then they get paired with a setter key that sets the aforementioned register to the word that has been spelled out.

- Below are the *alphabet letter* keys. There is one for each letter in the English language.

  - 🗝 Alphabet Letter ‘A’
  - 🗝 Alphabet Letter ‘B’
  - 🗝 Alphabet Letter ‘C’
  - 🗝 Alphabet Letter ‘D’
  - 🗝 Alphabet Letter ‘E’
  - 🗝 Alphabet Letter ‘F’
  - 🗝 Alphabet Letter ‘G’
  - 🗝 Alphabet Letter ‘H’
  - 🗝 Alphabet Letter ‘I’
  - 🗝 Alphabet Letter ‘J’
  - 🗝 Alphabet Letter ‘K’
  - 🗝 Alphabet Letter ‘L’
  - 🗝 Alphabet Letter ‘M’
  - 🗝 Alphabet Letter ‘N’
  - 🗝 Alphabet Letter ‘O’
  - 🗝 Alphabet Letter ‘P’

- Alphabet Letter ‘Q’
- Alphabet Letter ‘R’
- Alphabet Letter ‘S’
- Alphabet Letter ‘T’
- Alphabet Letter ‘U’
- Alphabet Letter ‘V’
- Alphabet Letter ‘W’
- Alphabet Letter ‘X’
- Alphabet Letter ‘Y’
- Alphabet Letter ‘Z’

Keyset: Word Search Related 🔑

- The *word search* keys are used in conjunction with the *alphabet letter* keys to form the sequence of keys by which the user can search for an instance of a particular word.

- When the user spells a word using the *alphabet letter* keys, they must then confirm the word that they have spelled using a number specifier. For example, if the user spelled the word 'displace' which is ambiguous regarding how it could technically be two different words ('displace' or 'place'), the user would need to specify that they are spelling 'displace' and not 'place' by using the number two ('2') to specify that the register should be set to the second word (out of those in the standard dictionary) back that is detected.

- If the user does not specify a number, the first word back is the one that the word for search register gets set to when the user calls the setter key.

- Below are three additional keys related to the discussed thus far in this section:

  - 🗝 Word Search Number One ('1')
  - 🗝 Word Search Number One ('2')
  - 🗝 Word Search Setter

- The word search technique follows suite with other applying keys regarding the two-consideration rule. That is, if the word for search register is set and then considered once by a sound-based capture key but not applied, the default behavior of the capture key is blocked. However, if the register value is considered for a second time and not applied once again, the default behavior of the capture key is not blocked this time, and the value of the register gets reset to NULL (nothing).

- There is another key that is used regarding the word search technique that allows the user to automatically reset the word for search register which is useful if the user wants to override the first-consideration-blocking effect of a non-dictionary-stored word (or compilation of letters) they have spelled. The key clears the word for search register if and only if (e.g. after it is called) the user subsequently calls a sound-based

capture key before they call any letter key. This particular constraint is in place for situations where the user may have called a sequence of letter keys and then used them to set the register (which is computationally expensive to begin with) and then accidentally called the key that clears the register. For the user to call an applying key next signals that he or she is more likely to have meant to called the clear key rather than on accident. To call a letter key signals a vice versa intention.

🗝 Word Search Register Clear

❖ One more key is introduced in this section which is used to copy a word that has been captured or otherwise pointed to using the 'blind point capture' key described on page X. This key copies the most recently referenced sound-based event in a given timeline (the most recent capture or otherwise the most recent word that the user blindly pointed to) into the word for search register if and only if the register is NULL when this key gets called.

🗝 Copy To Word Search Register

Register: Word For Search 💾

Register name: Word For Search; Data type: String; Possible Values: *Any dictionary word*

- The Word For Search register is used to keep track of which word will be captured when the user pairs it with a sound-based capture key (with or without additionally pairing it with *counting* number keys or *instance* number keys: if without, the value of the register pairs with the applying *capturing* key to search for the most recent instance of the specified word).

Keyset: Marking Related 🔑

- The *marking related* keys are used in conjunction with the *BOCT driving* keys to specify which particular captures should be moved by a particular *BOCT driving* key.
- If at least one capture within the BOCT is marked, all captures that are marked are eligible to be moved by the next (applied) *BOCT driving* key, and all captures that are unmarked are ineligible to be moved by that same (next) BOCT driving key.
- Intuitively, for public movement, marking can happen either before the public *BOCT driving* key is called or after (but before the aura reference). For private *BOCT driving* keys, the marking process must take place before that particular key is called.
- Captures that have been marked can be unmarked. If all captures get unmarked after one or more have been marked, then the process of driving the BOCT through auras returns to normal.
- There are two number keys within this keyset that are used to create a value that corresponds to the correct capture where the value one (‘1’) corresponds to the most recent, non-channeled capture (of any type), two (‘2’) corresponds to the second most recent, non-channeled capture, and so on. These numbers may get paired with arithmetic to form higher values.
  - 🗝 Marking Number One (‘1’)
  - 🗝 Marking Number Two (‘2’)
- To apply the number called or value created involving the *marking* number keys, there are two additional keys which mark and unmark captures respectively.
  - 🗝 Mark
  - 🗝 Unmark

Keyset: Reference Point Related 🔑

- The *reference point related* keys are used to create and utilize reference points which are WaveLight construct ‘points in time’ that the user adds with a creation key.

- The individually added reference points act as distinguishable events themselves because that is essentially what they are. The differences between reference points and other distinguishable events are that reference points are added with a called key (unlike any other distinguishable event) and also reference points are not captured, only referenced.

- There is only one use case for reference points which is to use them to start or complete (but not both) a link between two moments that is formed using the ‘Moment Link’ key. For example, a timeline might begin to unfold with distinguishable events ‘A’, ‘B’, and ‘C’, and the user might want to capture everything in between event ‘A’ and some moment in time that came after event ‘C’ (such as a sound that was not a word, like an instrument). If there were no distinguishable events to proceed event ‘C’ but the user knew that everything they wished to capture had already occurred, the user would either need to deliberately create a distinguishable event (e.g. say a word) or otherwise wait for one to occur (so that they can link ‘A’ with the distinguishable event that would have to proceed ‘C’). In either case, the situation is not ideal because it stalls movement and direction. For that reason, this set of keys is there so as to allow the user to insert a point of reference quickly and on demand.

- There are two reference point related keys: one to insert the reference point and one to seek (or reference) them. Creating reference points is intuitive, the user simply calls the creation key. Referencing them is also simple. The user can use counting number keys or instance number keys. With counting number keys, the user counts back in total reference points created. With instance number keys, the user can simply use the appropriate instance number or value and then follow with the seek key to reference the points based on the zero-point-at-BOCT-prime numbering convention.

- Only one number can be applied using a reference point seek key at once. The most recently called or formed appliable value (e.g. from the two types) will be applied to

the reference point seek key. The seek key will look at the most recently called or created number or value first. If it applies that one, then it does not consider any other numbers. If it does not apply the first one (or any preceding one) that it finds, any key that it considers but does not apply one is once-considered by the seek key and follows suite with other number keys regarding the twice-considered rule.

❖ Here are the two keys:

- 🗝 Reference Point Creator
- 🗝 Reference Point Seek

Keyset: Timestamp Related 🔑

- The *timestamp related* keys are used to create and utilize timestamps that are required to apply the five non-standalone capturing keys.

- The first key in this set is the one that is used to create timestamps.

  🗝 Moment Link: Links two captures or a capture and a single reference point to form a timestamp byproduct that extends between the two points in time.

- The ‘Moment Link’ key is paired between two captures in one of two ways. The first way involves a process by which the user makes a capture (or has already done so) then calls the ‘Moment Link’ key and then makes a second capture. Upon detection, the ‘Moment Link’ key links the most recent capture (out of those within the BOCT) to be made before it with the most recent capture to be made after it. If there are no captures within the BOCT when the ‘Moment Link’ key is detected, the ‘Moment Link’ key is immediately deleted. The ‘Moment Link’ key binds the capture before it with the next capture made by default unless the user calls the ‘Moment Link delay’ key which prevents one proceeding capturing key from completing the link, after which the subsequent capturing key will complete the link (unless the delay key is called again beforehand).

- The other way that the ‘Moment Link’ key gets paired with two captures is by using the recent capture keys which allows the user to manually bind the ‘Moment Link’ key with two specific captures rather than using the default function. Using this method, the user starts by calling a number or creating a value with the *recent capture* number keys and then from there, the user applies the number using the ‘Moment Link’ key which binds the number or value (only one- the last one called or created gets applied and all preceding ones get immediately deleted) the that ‘Moment Link’ key. From there, the user can repeat the process of calling a number or creating a value using the recent capture keys and after the user applies the ‘Moment Link’ key for a second time, the timestamp will be created between the two referenced captures.

- The user can use the on-the-fly linking method interchangeably with the manual binding method regarding any linking technique. The main consideration is whether a previous link has begun to be formed but has not been completed. A link that has been

started but has not finished must be completed prior to a second link being started. If a particular instance of the 'Moment Link' key being called is not capable of producing a timestamp due to not adhering to these constraints, that called key gets deleted.

- The user may link a capture with a reference point using 'Moment Link' by replacing the process by which the user would normally complete the creation of a timestamp by binding the 'Moment Link' key with the second of the two specified captures it would otherwise be linked to with a reference to a reference point. That is, a reference point can only be used to complete a timestamp, never to start one.

- After a timestamp has been created, it can be referenced so as to be applied using a specific capture key with the 'Tower Refract' key. The 'Tower Refract' key pairs with timestamps and capturing keys to apply a given timestamp to the next two capturing keys that are called (even if same key is called twice in a row). For example, let's say that the sentence "Hello, I hope you are well." was spoken aloud. If the user wished to capture the entirety of the sentence, an efficient way might involve capturing the word 'Hello' first, linking it with the word 'well' with 'Moment Link', and then proceeding to call the 'Tower Refract' key so as to apply the timestamp byproduct of the link between the two captures to the same channel, effectively capturing the entire sentence and overriding the two individual captures into a new, single capture including all six words. After the user calls the 'Tower Refract' key, they would simply need to call the (sound-based) capturing key that captures from the channel wherein the sentence was spoken. In doing so, the timestamp which in this situation spans between (inclusively) the word 'Hello' and 'well' gets applied to that specified channel and that is how the entire sentence gets captured.

- When a timestamp is created using the 'Moment Link' key, it has two **layers**. The first layer of a timestamp (typically) denotes the first reference to it, and the second layer denotes the second reference to it. A timestamp can be referenced two times using the 'Tower Refract' key. There is an exception to the idea that the first layer denotes the first reference of a timestamp always. The layer is more like a position for a reference to point, kind of like a seat on a vehicle. For example, cars typically have two front seats. This is analogous to the two layers of a timestamp. By default, the first reference to a timestamp will be placed in (or made to) the first layer however, there are specialty keys that allow the user to alter the default application function regarding capturing keys (e.g. references to timestamps) being related to specific layers of a timestamp.

- Prior to applying a capture key after a timestamp has been created and the 'Tower Refract' key has been called, the user may use timestamp layer blocking number keys

to form a number that denotes the number of layers that they wish for the next 'Tower Refract'-paired capturing key to skip where the number notation for the levels that it skips is ordered in such a way that the number one ('1') created with the timestamp layer blocking number keys would skip one layer starting with the first layer of the first timestamp to be created, the number two ('2') would skip two layers starting with the first layer of the first timestamp followed by the second layer of the first timestamp to be created, the number three ('3') would skip three layers- the third of which would be the first layer of the second timestamp to be created (assuming there are at least two timestamps created and neither have been referenced at all yet).

- When the user calls multiple timestamp layer blocking numbers or creates multiple values of that type (or if you will, one of each), only the most recently called or created one gets applied. The application occurs at the moment that an eligible capture key which would otherwise apply the next non-referenced layer of a timestamp gets called. For example, if there were two timestamps and neither the former nor the latter had been referenced at all, if the user called the 'Tower Refract' key then called or created the value two ('2') using timestamp layer blocking number then proceeded to call a capturing key, the capturing key would apply to layer one of the latter timestamp, rather than layer one of the former one, which it would do by default.

- Here are the six remaining keys which have been described in this section:

  - Moment Link Delay
  - Recent Capture Number One ('1')
  - Recent Capture Number Two ('2')
  - Tower Refract
  - Timestamp Layer Blocking Number One ('1')
  - Timestamp Layer Blocking Number Two ('2')

Keyset: Tower Refract Related 🔑

- The *Tower Refract related* keys are used to manipulate the default behavior of the 'Tower Refract' key.

- The first of the specialty functions of the 'Tower Refract' key is the ability to manipulate the layer targeting function thereof. Normally, the 'Tower Refract' key targets the layers starting from layer one of the first timestamp to be created and so on (as described in the last section). Using the layer targeting keys with the 'Tower Refract' key is slightly different than just blocking the layer from being referenced. Rather, the layer targeting keys lock the layer (number wise) that the 'Tower Refract' key will reference when applied with a capturing key. In a sense, it is another filtering technique regarding which layer or which timestamp a capturing key will be applied to. If a certain layer blocking number 'X' has been created, that number does not interfere or even interact with the layer targeting setting (e.g. layer one- meaning the 'Tower Refract' key will only target layer one of relevant timestamps). That is, when a 'Tower Refract' key is applied, it still skips over at least 'X' timestamp layers and then from there, it begins to apply to the next layer (e.g. layer one) of the first timestamp created out of the considered ones (regarding the layer blocking).

- The process of targeting specific layers of timestamps is especially useful when utilizing the Tower Refract looping numbers to cause the 'Tower Refract' key to apply a particular capture key to multiple layers from one or more timestamps.

- Take our same example from the last section: "Hello, I hope you are well." Imagine that the user had gone through the same process as before and had captured the words 'Hello' and 'well' and had formed a timestamp that spanned the two words (inclusively) in the process. Let's say that before the user called the 'Tower Refract' key and applied a capturing key with it, that there was another sentence spoken: "Today is going to be great." Imagine that the user wanted to capture all of the words in that sentence too, so prior to calling the 'Tower Refract' key at all, they went ahead and captured the words 'Today' and 'great' and in the process, created a timestamp that spanned the two words (inclusively) by linking them with 'Moment Link'. At this point, the user has two timestamps, neither or which have been referenced even once. From here, the user could efficiently make the two timestamp-applying captures that they want to make with a single 'Tower Refract' key paired with a single capturing key. For example, the user could call the 'Tower

Refract layer one lock' key followed by the 'Tower Refract looping number two' and then when the call and apply a subsequent 'Tower Refract' key, the first layer of both timestamps will be applied with the single capturing key.

- The last called eligible looping number is the only one that gets applied when a 'Tower Refract' key is applied. In cases where there are no timestamps with the specified layer available to reference or when the looping number is higher than the number of times that the 'Tower Refract' key can be applied when prioritizing layer target locking for a particular 'Tower Refract' key, both the layer lock and the looping number are deleted. The layer lock (register) can be reset manually with the 'Tower Refract layer lock reset' key.

- Everything that has been discussed within this section and the preceding one regarding manipulation of the default behavior of the 'Tower Refract' key being applied to timestamps has be solely with regard to a particular instance of the 'Tower Refract' key being called and applied. For example, if a layer is blocked from being referenced when a 'Tower Refract' key is applied, the blocking mechanism is immediately deleted thereafter. There is a way to permanently delete the layers of timestamps which is to call both the 'Delete blocked timestamp layers' key as well as the 'Confirm delete timestamp layers' key prior to applying a 'Tower Refract' key that upon application, skips one or more layers of one or more timestamps due to the two aforementioned techniques being used. When both of these keys are called and then a 'Tower Refract' key gets applied, any skipped layer from any referenced timestamp gets permanently deleted. If only one of these two keys is applied an/ or there are no skipped rows upon application of the subsequent 'Tower Refract' key, the timestamp layer deleting key(s) get(s) deleted.

  - Tower Refract Layer One ('1') Lock
  - Tower Refract Layer Two ('2') Lock
  - Tower Refract Layer Reset
  - Tower Refract Loop Number One ('1')
  - Tower Refract Loop Number Two ('2')
  - Delete Blocked Timestamp Layers
  - Confirm Delete Timestamp Layers

Timestamp Constraints 

- There are a few additional details about timestamp creation and referencing that are worth mentioning.

- First of all, when timestamps are created, there is a particularity regarding the linking process that might be relevant to users at times. When a link is made, the individual capture or captures involved in the link are either referenced regarding their Began or Ended field (from the 'BOCT' table), but not both date/time values are used.

- Let's use an intuitive example. Imagine the sentence, "How is the weather today?" was spoken and that the user began by capturing the word in the middle thereof: 'the'. The distinction regarding whether a particular capture will be referenced by its Began or Ended times is related to the direction of the link that the user creates in the process of using it. For example, if the user used the 'Moment Link' key after capturing the word 'the' and then proceeded to subsequently capture the word 'How', then the timestamp created would span between the start of the word 'How' until the end of the word 'the'. If instead, the user had proceeded to capture the word 'today' rather than the word 'How', then the timestamp would span between the words 'the' and 'today' and therefore, the timestamp would reference the start of the word 'the' and the end of the word 'today'. If this logic is not intuitive enough, just know that the distinction is related to the fact that timestamps target events that always take place in a sequential manner (relative to time), however the capture process can be reversed in that a preceding capture can possibly target something that happened later in time than that of a capture that was made subsequently for something that happened earlier in time.

- The next detail about timestamps is definitely more relevant to the user's focus when working with timestamps. This detail relates to the fact that the 'Tower Refract' key does not block the function of other numbers when it is paired with an applying capture key. For example, if we take the same sentence from above (e.g. the last paragraph) and we utilize the scenario where the user began by capturing the word 'the' and then proceeded to link it with the word 'How', we can define a helpful example. Let's say the user was this far in the sequence- with the two captures made and the timestamp spanning between them created. If the user wanted to capture every word in the sentence, they could do so by calling the 'Tower Refract' key and then subsequently, 'counting number two', 'counting

number one', and then finally the 'sound (*based) capture' key. In doing so, the user is able to simultaneously apply the timestamp to the capturing key and additionally, apply the number keys using the capture key.

- Obviously, the application of the timestamp using the 'Tower Refract' key is not showing its true power due to the fact that is only capturing one additional word ('is') however, there is something very interesting worth noting occurring in this example. When the user applies 'Tower Refract' as well as the number keys using the same capturing key in such a way that a period of time within a particular channel without any breaks in between is captured, it creates a single, merged capture. That is, because the three captures that would result from the single capturing key being applied in this example involve a seamless coverage of the sensory input/ output stream regarding a particular channel, the captures get automatically merged within the BOCT to form a single capture spanning the entire sentence- from the word 'How' to the word 'today' (inclusively). That might not be as interesting as it was chalked up to be- but efficiency wise, it is a great thing.

Register: Tower Refract Layer Lock 📋

Register name: Tower Refract Layer Lock; Data type: String; Possible Values: NULL, One, Two

- The Tower Refract layer lock register is used to keep track of which layer regarding subsequently referenced timestamps that the 'Tower Refract' key will target.
- When this register is set to 'One', only the first layer of eligible timestamps will be referenced by the next 'Tower Refract' key to be applied, and vice versa if the register is set to 'Two'.
- If the register is already set, it can be set again to the other of the two non-null settings and this can continue (e.g. back and forth) between applications of the register. That is, if the register is set to 'One' (for example) already, calling 'Timestamp layer lock two' sets the register to 'Two' without it ever becoming null.
- If this register is set (e.g. not NULL), it gets reset after the next 'Tower Refract' key is applied. Again, the user can manually reset the register (to NULL) with the 'Tower Refract layer target reset' key defined on page X.

Register: Tower Refract Loop Number 📋

Register name: Tower Refract Loop Number; Data type: String; Possible Values: *Numbers

- The Tower Refract loop number register is used to keep track of the number of times that the next 'Tower Refract' that gets applied will apply (sequentially to eligible timestamp layers).

- This register gets set automatically any time that a relevant number is created using the *Tower Refract looping* number keys. Again, only the most recently created of those numbers is applied. Any time a new number of that type is created, the previous ones get disabled and then deleted upon application of any different number of that type. A previous number or value can be re-enabled by disabling subsequent, non-applied ones however, any single number or value resulting from an equation that has been disabled cannot be altered using arithmetic.

- Applying a 'Tower Refract' key resets this register to NULL.

Keyset: Aura Deriving 🔑

- The *aura deriving* keys are used to derive captures from auras which they have been channeled into.

- The aura-deriving process is very similar (almost identically) to the BOCT-driving process except it occurs in a reversed fashion. After a capture or captures are channeled into an aura, one or more can be derived from that aura and placed back into the BOCT (with a new capture ID- as if a new capture were made).

- As stated, the process is similar to BOCT-driving. Actually, aura-deriving *is* BOCT-driving but instead of the captures moving from the BOCT to the aura, they move in the opposite direction. The user starts by calling a derive from light/ dark key which specifies which side of the aura the derivation will occur from. From there, the user simply references the relevant aura they wish to derive from. If there is anything in that particular side of the aura that gets referenced, everything within the aura will transfer from it back into the BOCT.

- The aura deriving keyset includes two number keys which are used to form one or more numbers or values which will be applied to the derivation process in such a way that only the captures within an aura that correspond to those numbers in terms of recency will be derived (rather than all of them).

- The four aura deriving keys that are used to specify which side of an aura the derivation process will occur with regard to (two keys for each side) operate on a polarity basis just like the *BOCT driving* keys and similarly, they are respectively limited to moving a particular capture a single time.

- In general, the aura deriving process is intuitive and straightforward. Here are the keys in this set:

  - 🗝 Derive From Light One
  - 🗝 Derive From Light Two
  - 🗝 Derive From Dark One
  - 🗝 Derive From Dark Two

- Deriving Number One (‘1’)
- Deriving Number Two (‘2’)

Aura Deriving Constraints 

- When the BOCT-driving process is used to move captures into auras, they are ordered in the ‘Auras’ table the same way that they were ordered in the BOCT and by default, aura deriving derives all captures from a particular aura. If there are three captures in a particular aura and the user creates the values two (‘2’) and three (‘3’) using the *aura deriving* number keys paired with arithmetic then subsequently derives from an aura, only the second and third-most recent captures (in terms of when they were captured) that entered that aura will be derived from it. You may notice the similarity between aura deriving numbers and BOCT marking. Their function is the same, however there is one fewer step involved in aura deriving in that the user need not apply the numbers prior to making the aura reference (as they must with BOCT marking) because the reference does that automatically in this case.

- If a user leveraged the secondary acceptance quality of a particular aura so as to add a secondarily-accepted capture type into said aura and then goes on to derive all of the primarily accepted capture types from that aura, the secondarily accepted type of capture is automatically derived along with the last primarily accepted type of capture that gets derived from the aura.

- The process of deriving captures from an aura never merges them even if multiple captures which were split during the process of transferring them from the BOCT into one or more auras- only the capturing process itself or the movement of captures from the BOCT to auras can merge what would otherwise be multiple captures into one.

Register: Derive From Light 📋

Register name: Derive From Light; Data type: String; Possible values: NULL, ‘One’, ‘Two’

- The Derive from light register is used to keep track of which of the two *aura deriving* (from light) keys needs to be used next in terms of alternating for polarity sake.
- This register gets set automatically based on the use of the two ‘derive from light’ keys.
- The polarity constraint of *aura deriving* (from light) keys can be overridden in situations where it would otherwise limit the use thereof however, the ability for an aura deriving key to move a capture more than one time cannot be overridden, just as is the case with *BOCT-driving* keys.

Register: Derive From Dark 📋

Register name: Derive From Dark; Data type: String; Possible values: NULL, ‘One’, ‘Two’

- The Derive from dark register is used to keep track of which of the two *aura deriving* (from dark) keys needs to be used next in terms of alternating for polarity sake.

- This register gets set automatically based on the use of the two ‘derive from dark’ keys.

- The polarity constraint of *aura deriving* (from dark) keys can be overridden in situations where it would otherwise limit the use thereof however, the ability for an aura deriving key to move a capture more than one time cannot be overridden, just as is the case with *BOCT-driving* keys.

Keyset: Aura Moving 🔑

- The *aura moving* keys are used to move auras to and from other WaveLight users.

- Moving an aura to or from a person is an action that indicates to that person that you want their attention to follow a certain train of thought or to take a certain stance relative to some event in the sensory input/ output stream. For example, if you channel a capture involving a specific word into an aura and then move it to someone, it may indicate that you wish for them to focus on that word. The nature of the specific actions can vary but all in all, moving auras gives WaveLight users a lot of control regarding the connotation of their communication and expression of attention to detail.

- An aura can be moved either before or after it contains (a) capture(s). The reason that BOCT driving is described as either public or private is that public movement is generally thought of as being viewable to other users in a public sense whereas moving captures into auras after the aura has already been moved to a specific person (private BOCT driving) is more discreet regarding the way that only that person can know what you are channeling into that aura.

- Aura movement keys involve a direction (of movement) key and then a subsequent aura reference. After the user calls the direction-based key, the user can apply the relevant function by subsequently referencing the aura they desire to move that way.

- Aura moving keys operate on a polarity basis which means that there are twin keys within this keyset (two for the move-to and move-from functions respectively). The aura moving keys also follow a sequential-order-based movement polarity constraint that is slightly more complex than what is observed with the BOCT driving keys. That is, not only do the two types of aura movement keys have to alternate simply based on the key-calling polarity constraint when called in a consecutive manner, the alternation itself must align with the order that the auras were moved in with respect to the order they were created in. For example, if three blue auras were created and then the first and third ones were moved with ‘move to one’ and ‘move to two’ respectively, then if the user tried to move the second aura with the same type of movement (move-to), there would be a conflict regarding how the movement polarity of the auras does not align with either key. This situation may

seem confusing, however it will be discussed more thoroughly in the next section. For now, just know that there is an additional key that is used to bridge this polarity concern by applying both movement keys of a specific type to a single reference (similar to the BOCT 'lock-in' rule override).

- Aura movement is generally very simple. The main thing to keep in mind is that moving a containing aura or channeling into a moved aura is final meaning that that aura cannot be moved, moved back, channeled into, or derived from again.

- Here are the six aura movement keys in this set:

  - Move To One ('1')
  - Move To Two ('2')
  - Move Back
  - Move From One ('1')
  - Move From Two ('2')
  - Move And

Aura Moving Constraints 

- The *aura movement* keys follow a few constraints that are necessary to adhere to in order to achieve the desired functionality thereof.

- As mentioned in the last section, moving an aura that is containing or channeling captures into an already moved aura is final meaning the aura cannot be moved again nor channeled into after that.

- For an aura to move *to* or *from* another WaveLight user, the aura must already be containing or there otherwise must exist within the BOCT a capture or captures which activate the aura upon its movement. For example, if (immediately after moving an aura) a called *private BOCT driving* key would not channel anything into an aura that the user wants to move, the aura cannot move. This is a momentary check and even if there exists a capture or capture within the BOCT that could be channeled into the aura but they are not marked and one or more other captures which cannot be channeled into the aura are also in the BOCT and are marked, the aura still cannot move the aura.

- The only non-containing aura that a user can move *from* another user to themself is the pink aura which channels decisions because decisions are the only capture type that the user can privately channel from another user to themself. Decision captures (when shown via the light of an aura) indicate to others what the user intends to do with regard to some decision. When a user channels a decision into an aura's light side and then moves the aura to someone else (or when the order is reversed and they channel into the aura after it is moved), this indicates to that other person that the user intends to decide against that decision. On the contrary, if the user moves an aura that contains a decision capture in its light side from someone else to themselves or if they channel into the light of an aura after moving it from someone to themself, that indicates that the user intends to decide in favor of the decision.

- The movement polarity constraint introduced in the last section is a constraint that prevents aura movement from deviating from alternation-based cycling. This constraint only applies to sets of three or more auras which were created consecutively (back to back) which the user wishes to move all the same way.

- For example, imagine the user created three auras in a row (of any color): ‘A’, ‘B’, and ‘C’. If the user planned to move all three auras *to* (e.g. rather than *from*) any other user, they would need to ensure that the movement of the auras follows the movement polarity constraint which states that the movement keys used to move consecutive auras in the same way must alternate back and forth between the two of that type. For example, the user could use ‘move to one’, ‘move to two’, and ‘move to one’ to move auras ‘A’, ‘B’, and ‘C’ respectively. Keep in mind that the order that the auras get moved does not matter, only that the movement polarity alternates between consecutive auras regarding when they were created. For example, the user could instead begin by using ‘move to two’ to move aura ‘B’ then proceed to move aura ‘A’ with ‘move to one’ and then finally move aura ‘C’ with ‘move to one’ paired with the polarity override key. All that matters with this constraint is that when the auras are moved, the two keys from the pair with the function that moved them alternate. Also, if there is any break in the movement type of an aura (e.g. switching between moving *to* to moving *from*), the constraint resets as soon as the first movement of a new type occurs.

- The movement polarity constraint only applies if an aura is not closed. For example, if the user created two auras (‘C’ and ‘D’) and then moved aura ‘C’ with ‘move to one’ then channeled into it with a private BOCT driving key (thus closing the aura), the user could proceed to move aura ‘D’ with ‘move to one’ (assuming they adhere to the key-calling polarity constraint).

- If a user moves an aura and then proceeds to move it back, the movement polarity constraint does not restrict whether the user can move the aura again, however the aura movement keys themselves (excluding the ‘move back’ key) can only move a particular aura one time per key, and that constraint could create a conflict. For example, take our three auras from before (‘A’, ‘B’, and ‘C’). Imagine that the user began by moving auras ‘A’ and ‘C’ with ‘move to one’ followed by ‘move to one’ again paired with the polarity override key. Imagine that they proceeded to move aura ‘B’ with ‘move to two’ then moved aura ‘B’ back with the ‘move back’ key. Now, the user cannot move aura ‘B’ with ‘move to two’ a second time due to the one-movement-per-key-constraint and also, the user cannot move aura ‘B’ with ‘move to one’ (even if the key-calling polarity constraint is adhered to) due to the aura movement polarity constraint. For that reason, there exists a specialty key called ‘move and’ which allows the user to pair both keys in the pair of a particular type (the ‘one’ and ‘two’ keys from the pair creates the name *and*) to move auras like aura ‘B’ in this example when it would otherwise require moving other auras back using the ‘move back’ key (or may not even be possible to do so).

- The ‘move and’ key is called in a similar way to that which the arithmetic operators keys are called. The key must be called in between calling the two keys in a pair regarding a certain aura movement type (the two keys in the pair can be called in either order) and then when an eligible aura is referenced, both the aura movement polarity constraint as well as the key-calling constraint are bypassed and the aura can move this way. However, if the user moves the aura back afterward, the user cannot use this bypass again and if that was done in a situation like our example above, the user would have to proceed to move auras ‘A’ and ‘C’ back with ‘move back’ then they could move aura ‘B’ using ‘move to one’. It is possible that an aura could become impossible to move if the user exhausts all possibilities.

- If multiple aura movement keys are called and then a single aura reference is made, the first function (i.e. movement type) that was triggered out of all the actively triggered movement types is the one that applies (similar to what was discussed in the BOCT driving constraints section).

Register: Move To Polarity 📋

Register name: Move To Polarity; Data type: String; Possible Values: NULL, ‘One’, ‘Two’

- The move to polarity register is used to keep track of the key-calling polarity of the ‘move to one’ and ‘move to two’ aura-moving keys.

- The register gets automatically set depending on how the user calls the two keys in the pair and it gets reset when the BOCT becomes prime.

Register: Move From Polarity 📋

Register name: Move From Polarity; Data type: String; Possible Values: NULL, ‘One’, ‘Two’

- The move from polarity register is used to keep track of the key-calling polarity of the ‘move from one’ and ‘move from two’ aura-moving keys.
- The register gets automatically set depending on how the user calls the two keys in the pair and it gets reset when the BOCT becomes prime.

Keyset: i-notation Related 🔑

❖ The i-notation related keys are used to form values relative to the *i'th* instance of some distinguishable event.

❖ Below are the i-notation keys.

- 🗝 Last ('i'): Targets the most recent (e.g. last) instance of some specified object.
- 🗝 Reverse Skip ('i'-1): Targets the second to last instance of some specified object.
- 🗝 Echo Subset (1-('i'-1)): Targets the first through the second to last instance of some specified object.
- 🗝 Echo Set (1-'i'): Targets all instances of some specified object.

❖ The i-notation keys are able to be paired with instance numbers (e.g. to form a value relative to 'i') or may operate on their and be paired with these types of capture keys:

- All sound-based capture keys (when using word search technique)
- Person thought capture

❖ The i-notation keys are also able to be paired with key component keys to target key components to enable or disable them. To do so, the user may utilize the 'disable next called key' key then subsequently, immediately call the i-notation key or several of those keys in order to link them.

- 🗝 Disable Next Called Key: Automatically disables the next called key. The key that gets called subsequently never begins its function. It simply gets placed in the sequence disabled from the start. The key can be enabled later on.

❖ If the user calls the 'disable next called key' key then immediately calls an i-notation key without calling any other type of key, the i-notation key gets linked to that disabled key. For example, the user may want to disable all instances of the 'counting number one' key which may have been called several times since the BOCT became prime. Thus, the user could call 'disable next called key' then call 'Echo Subset' or 'Echo Set'

which would point to all of the instances of that key up to the second to last one or all up to and including the last one respectively and then the user could subsequently use the ‘key component disable’ key to disable all of them (keep in mind that disabling an already disabled key is legal, though it does alter anything).

- There can be more than one i-notation number created at a time. For example, if the user had spoken the word ‘hello’ twice since the BOCT became prime then set the word for search register to ‘hello’ then called ‘Last’ and ‘Reverse Skip’, both instances of the word could be captured that way.

Keyset: Pointer Movement Related 🔑

- The *pointer movement related* keys are used to move a 'present pointer' related to what is captured or will be next from within the sensory input/ output timeline.

- Any time a capture from within one of the eight standalone capturing keys is made using either the capture key's default behavior or by pairing a number with the capture key, the respective present pointer register for that channel (there are eight- one for each channel), is set to that most recent capture's Capture ID.

- There are keys in this set that allow the user to move the present pointer register for a given channel without immediately capturing the thing it is moved to. For example, the sentence "How is everybody doing today?" may have been spoken and the user may wish to capture the word 'everybody' by using a sound-based capturing key. If the user wanted to make the capture by using the word-search technique described in an earlier section, they could do so in an efficient way by leveraging a special key which will be listed later in this section. Instead of spelling out the word 'everybody' with all nine letters, the user could spell the word 'is' using only two and then call the 'blind point capture' key prior to calling the relevant sound-based capturing key (which normally would capture the word 'is'). By doing so, the user has effectively moved the 'present pointer' of that specific channel (e.g. one of the sound channels) to the word 'is' without capturing it. From there, the user can call a second specialty key listed in this section called a step key and then proceed to call the relevant capturing key and in the process the user will not only move the present pointer register one word forward to the word 'is', but will capture it simultaneously.

- The 'step channel set' key is used to manipulate the channel that a proceeding step key will apply to. By default, the step channel register is set to the channel (of the eight that contain distinguishable events- the only channels wherein the user can make a step-capture) that the most recently made capture that still resides in the BOCT was made from. The user can call this key followed by a step key and then one of the eight standalone capturing keys (which can be disabled using the 'disable next capture' key so that that instance of calling the key does not itself make a capture) to have the step channel register set to a different channel. That is, the only way to have a step key apply to a particular capture key is to have the step channel register match the capture key (which is achieved automatically any time a capture is made- the step channel register gets set to that channel). If there are no previous captures

to step from (e.g. in the BOCT or as a blind pointer (blind pointers are defined on page X) from) within the channel that the user specifies with this technique, the capture key defaults to one back and both the 'step channel set' and the step key that was used are deleted.

- The two step keys default to stepping by one distinguishable event unless otherwise paired with a counting number in which case, the step key may only apply the most recently called or produced (appliable) number or value of the counting type. If there are more than one counting number or value present when the step key is applied, any that precede the most recently called appliable one (e.g. the one that gets applied) will have not be considered and any from the most recently called one total to the one called subsequently to the one that gets applied (i.e. non-appliable numbers or values of an eligible type, if applicable) that get considered but remain non-applied follow suite with the two-considerations-without-application-to-delete rule defined previously for counting numbers. Likewise, if one or more counting numbers or values exist and a step key is called but the step key does not apply any of them, the step key's default behavior is blocked unless any of those non-appliable counting numbers was considered for a second time in this process.

- The user may trigger both a step key as well as the 'blind point capture' key and then proceed to call a standalone capturing key to incrementally step forward or backward in that channel without capturing what is stepped to.

- The user may reset the present pointer register for a specific channel by calling the 'present pointer reset' key and the proceeding to either (1) call the 'blind point capture' key or (2) call the 'disable next called key' (as defined on page X) and then in either situation, finish by calling the specific capture key that captures from the channel the user wishes to reset the present pointer register for. Resetting the register will set the value back to the capture ID of the most recent capture from that channel which is what happens by default when a normal (non-pointer-manipulated) capture is made within a given channel.

- Here are the five keys in this set that have been alluded to in this section:

  - Blind Point Capture
  - Step Channel Set
  - Step Forward
  - Step Backward

 Present Pointer Reset

---

Register set: Present Pointer 💾

Register set: Present Pointer; Data type: Integer; Possible Values: NULL, Blind Point IDs

- The Present Pointer register set is a set of registers used to keep track of which present pointer from which channel is pointing to a non-captured thing, and what it is pointing to.

- Each channel regarding the eight distinguishable event-capturing keys and the respective channels they each capture from has its own individual present pointer register (with the exception that the 'sound in real life' and 'sound in real life from self' capture keys share one register: sound in real life present pointer) which is either NULL or is set to a number which indicates via a table reference what a given channel's present pointer is pointing to.

- The present pointer is used to keep track of what will be captured when the user uses the step key which again, is very useful when it helps the user avoid spelling out longer words by spelling shorter ones, as described on page X.

- The present pointer is set by using the 'blind point capture' key and then calling a subsequent capture key. Whether the 'blind point capture' key has been called or not, the user must call a capture key (and any other appropriate keys when applicable) in order to stage the capture of whatever it is that they want to point to. The only difference is if the user has called the 'disable next capture' key, then the capture is blocked and only the present pointer for that channel is set.

- The 'blind point capture' and 'disable next called key' (defined on page X) keys are different. The former is used to set the present pointer for a specific channel (and also counts as a consideration regarding the two-considerations-without-application-to-delete rule for numbers) and thus, the subsequently called key performs some (albeit, limited) function however, the 'disable next called key' key entirely prevents the function of the subsequent key (which can later be enabled to performs some function according to the sequence up to that point) of any type. If the 'disable next capture' is called and then 'disable next called key' and then a capturing key immediately follow, the 'disable next capture' key has not yet been applied because a capture would not have been made even if it was not triggered. The key only applies in cases where a capture would have otherwise been made.

- The present pointer register for a given channel is set to a number that coincides with the Blind_Point_ID from the Blind Pointing table which keeps track of details related to the present pointer for that channel which again, is only set to a value other than NULL when the 'stepping point' (what will be stepped *from* if a step key is called) is not a capture.

- For example, imagine the user spoke the words "I fascinate the world" out loud and then many more words were spoken after that and then the user wanted to capture the word 'fascinate'. If the user called the 'blind point capture' key then (used the word search trick alluded to previously and) spelled the word 'I' and then used the appropriate sound-based capture key, the present pointer for that channel would 'blind point' to the word 'I' meaning that the 'stepping point' would then be set to that word however it would not be captured. Then the user could call 'step forward' and capture the word 'fascinate', at which point the present pointer for that specific sound channel would get set to NULL again. When a 'stepping point' is a capture, the BOCT table is sufficient for tracking the details thereof.

Table: Blind Pointers 🗂️

Field Name	⭐Blind_Point_ID	Setting_Key	Start	End
Data Type	Integer	Integer	Date/Time	Date/Time

- The Blind Pointers table is used to keep track of the status of present pointers. When the user makes a new blind point to something from the sensory input/ output stream and leading up to making a capture from within that same sensory channel, a single Blind_Point_ID is used to keep track of any changes to that blind pointer.

- For example, if the user spoke the sentence “Hello there, world” out loud and then ‘blind-pointed’ to the word ‘Hello’ after having had made a capture from the channel wherein this sentence was spoken and prior to having made a subsequent capture thereto, a new **blind pointer** would be created. If the user used ‘blind point capture’ and then ‘step forward’ after that, the same blind point (i.e. with the same ID) would be altered within that same row such that the setting key, start, and end would all change accordingly but a new blind pointer would not be created.

- If the user uses the present pointer for a particular channel directly to make a capture (through the use of a step key- the only way to do so), or otherwise makes a capture from a channel wherein they have created a blind pointer through a process that does not utilize the blind pointer itself (such as capturing the word ‘world’ from the example above by using the sound capture’s default behavior while the present pointer register for that channel was set to ‘Hello’), the blind pointer with that particular ID gets deleted from the table either way.

- Here are the details for the fields in this table:

  - Blind_Point_ID: A primary (unique) field that identifies any one given blind pointer.
  - Setting_Key_ID: References Called_Key_ID from the Called Keys table. Is used to identify the capturing key that would otherwise have made the capture which the ‘blind point capture’ blocked.

- Start: The start time in terms of the sensory input/ output stream that the blind pointer points to.
- End: The end time in terms of the sensory input/ output stream that the blind pointer points to.

Register: Step Channel 💾

Register set: Step Channel; Data type: String; Possible Values: The eight standalone channels

- The Step Channel register is used to keep track of the channel that the user will step from if a step key is used.
- The user sets this register in one of the two ways described on page X. After it is set, the next time that a step key is called and then a subsequent capture key that captures from the same channel that this register is set to, the step will be applied.
- For example, if the user set this register to 'sound in real life' and had a stepping point available (e.g. a BOCT capture or a blind pointer) from either of the real sound channels (due to the fact that the two real sound capturing keys capture from the same sensory input/ output stream, they are considered as one channel for stepping and blind pointing purposes) and then proceeded to call 'step forward' and then one of the real sound-based capture keys, the step key would apply.
- If the user calls the 'step channel set' key then makes a capture or otherwise blocks a capture upon calling a capturing key from one of the non-distinguishable-event-based capturing keys, then this register does not get altered.

Keyset: Capture Style Related 🔑

- The *capture style* related keys are used to temporarily alter the default capturing (overwrite) behavior programmed into the keys in the framework.

- As was described on page X, by default, “If capture ‘A’ is replaced by a second capture ‘B’ because the content of A is a subset of the content of B then capture A would get nullified or effectively deleted.”

- There is one key to disable this default behavior and there is another one to re-enable it. Disabling this default behavior not only prevents captures that are subsets of proceeding captures from being deleted, it also removes any overlap in terms of what is added to the BOCT after the subsequent capture is made.

- For example, if the sentence “I am getting tired” was spoken and the user had already captured the word ‘I’ and then they subsequently captured the entire sentence at once using ‘Tower Refract’ then by default, the capture of the single word would be removed from the BOCT and the latter capture would replace it however if the user disables this default behavior, then the subsequent capture would include only “am getting tired” even if the ‘Tower Refract’ had additionally considered the word ‘I’.

- Disabling this default behavior only applies to the next capture made and even if the function is not utilized due to an overwrite possibility not existing in the subsequent capture called prior to disabling the default behavior, the disabling key gets deleted.

  - 🗝 Disable Next Capture Overwrite
  - 🗝 Re-enable Next Capture Overwrite

Keyset: Flag Related 🔑

- The *flag related* keys are used to create flags which are useful for annotating content within auras.

- Flags within the WaveLight framework are a construct used by the user to sort of comment on what they have captured and channeled into a particular aura.

- A flag may be a single word, a true/ false value, an image, a snippet from a music video, or even an emoji. What the user can create to be a flag is fairly open-ended. The point of flags is to increase the accuracy of the emotional evocation the user intends to deliver with their personal style of capturing and channeling.

- To begin the flag creation process, the user must start by calling the 'flag work enter' key and then they will have to reference an aura that is either publicly containing or which is moved and activated then they may proceed to put flags into the aura.

- For activated, non-containing auras, flags do not enter the aura until the user receives the aura. If the user is doing flag work on an activated aura and then it becomes deactivated, all flags that would have gone into the aura (e.g. if the user moved the aura to another user) get deleted permanently.

- For example, if a blue aura 'B' was created and the user captured the word 'Hello' and then channeled the word into it publicly, then the user could call 'flag work enter' and then reference the aura (with the aura reference keys defined on page X). After that, the user can proceed to use Twiddler chords which get preset to certain specific flags (the same way that each key gets a chord, so does each of the flags). By simply 'calling' a flag, the user puts it into that aura.

- The user may remove flags from an aura using the *flag number* keys paried with a deletion key. The flags get put into the aura in a first-in-last-out order meaning that if three flags have been put into an aura, the first one to be put into it would be referenced with the number three ('3') using the *flag number* keys. The user deletes the flags using the 'flag delete' key which defaults to one back if it is not paired with a flag number.

- The user can ‘step out’ of flag work within a particular aura by using the ‘flag step out’ key. This way, the user can enter flag work with other auras. The user cannot begin working on the flags of a different aura until they step out of the flag work with a previous aura if they were doing flag work with one before. Closing an aura by having it contain a capture or captures while it is moved automatically steps the user out. The user cannot do flag work with closed auras.

  - Flag Work Enter
  - Flag Number One (‘1’)
  - Flag Number Two (‘2’)
  - Flag Delete
