ndscan
Search docs
CONTENTS:
ndscan.experiment API
Fragments
Scans
Experiment entry points
ndscan.results API
Coding conventions
Design retrospective

ndscan.experiment API

Experiment-side `ndscan` interface

`ndscan.experiment` contains the code for implementing the `ndscan` primitives from ARTIQ experiments (as opposed to submitting the experiments with certain parameters, or later analysing and plotting the generated data).

The top-level module provides a single convenient way to import commonly used symbols from experiment client code, like `artiq.experiment` does for upstream ARTIQ:

```
# Import commonly used symbols, including all of artiq.experiment:
from ndscan.experiment import *
```

Fragments

`ndscan.experiment.fragment` module

class **ndscan.experiment.fragment.Fragment**(*managers_or_parent*, **args*, ***kwargs*) [source]

Main building block.

build(*fragment_path: list[str]*, **args*, ***kwargs*) [source]

Initialise this fragment instance; called from the `HasEnvironment` constructor.

This sets up the machinery for registering parameters and result channels with the fragment tree, and then calls `build_fragment()` to actually perform the fragment-specific setup. This method should not typically be overwritten.

Parameters:
- **fragment_path** – Full path of the fragment, as a list starting from the root. For instance, `[]` for the top-level fragment, or `["foo", "bar"]` for a subfragment created by `setattr_fragment("bar", …)` in a fragment created by `setattr_fragment("foo", …)`.
- **args** – Arguments to be forwarded to `build_fragment()`.
- **kwargs** – Keyword arguments to be forwarded to `build_fragment()`.

host_setup() [source]

Perform host-side initialisation.

For fragments used as part of an on-core-device scan (i.e. with an `@kernel` `device_setup()` / `run_once()`), this will be called on the host, immediately before the top-level kernel function is entered.

Typical uses include initialising member variables for latter use in `@kernel` functions, and setting parameters not modifiable from kernels (e.g. because modifying a parameter requires launching another experiment to effect the change).

The default implementation calls `host_setup()` recursively on all subfragments. When overriding it in a fragment with subfragments, consider forwarding to the

default implementation (see example).

Example:

```
def host_setup(self):
    initialise_some_things()

    # To continue to initialise all subfragments, invoke the parent
    # implementation:
    super().host_setup()
```

device_setup() → None [source]

Perform core-device-side initialisation.

A typical implementation will make sure that any hardware state represented by the fragment (e.g. some DAC voltages, DDS frequencies, etc.) is updated to match the fragment parameters.

If the fragment is used as part of a scan, `device_setup()` will be called immediately before each `ExpFragment.run_once()` call (and, for on-core-device scans, from within the same kernel).

The default implementation calls `device_setup_subfragments()` to initialise all subfragments. When overriding it, consider forwarding to it too unless a special initialisation order, etc. is required (see example).

Example:

```
@kernel
def device_setup(self):
    self.device_setup_subfragments()

    self.core.break_realtime()
    if self.my_frequency.changed_after_use():
        self.my_dds.set(self.my_frequency.use())
    self.my_ttl.on()
```

device_setup_subfragments() → None [source]

Call `device_setup()` on all subfragments.

This is the default implementation for `device_setup()`, but is kept separate so that subfragments overriding `device_setup()` can still access it. (ARTIQ Python does not support calling superclass implementations in a polymorphic way – `Fragment.device_setup(self)` could be used from one subclass, but would cause the argument type to be inferred as that subclass. Only direct member function calls are special-cased to be generic on the *self* type.)

host_cleanup() [source]

Perform host-side cleanup after an experiment has been run.

This is the equivalent of `host_setup()` to be run *after* the main experiment. It is executed on the host after the top-level kernel function, if any, has been left, as control is about to leave the experiment (whether because a scan has been finished, or the experiment is about to be paused for a higher-priority run to be scheduled in.

Typically, fragments should strive to completely initialise the state of all their dependencies for robustness. As such, manual cleanup should almost never be necessary.

By default, calls *host_cleanup()* on all subfragments, in reverse initialisation order. The default implementation catches all exceptions thrown from cleanups and converts them into log messages to ensure no cleanups are skipped. As there will be no exception propagating to the caller to mark the experiment as failed, failing cleanups should be avoided.

Example:

```
def host_cleanup(self):
    tear_down_some_things()
    super().host_setup()
```

device_cleanup()→ None [source]

Perform core-device-side teardown.

This is the equivalent of `device_setup()`, run after the main experiment. It is executed on the core device every time the top-level kernel is about to be left.

Thus, if the fragment is used as part of a scan, `device_cleanup()` will be typically be called once at the end of each scan (while `device_setup()` will be called once per scan point).

The default implementation calls `device_clean_subfragments()` to clean up all subfragments in reverse initialisation order. When overriding it, consider forwarding to it too (see example).

Example:

```
@kernel
def device_cleanup(self):
    clean_up_some_things()
    self.device_cleanup_subfragments()
```

device_cleanup_subfragments()→ None [source]

Call `device_cleanup()` on all subfragments.

This is the default implementation for `device_cleanup()`, but is kept separate so that subfragments overriding `device_cleanup()` can still access it.

`device_cleanup()` is invoked on all subfragments in reverse initialisation order. To ensure no cleanups are skipped, any exceptions thrown from cleanups are caught and converted into log messages. As there will be no exception propagating to the caller to mark the experiment as failed, failing cleanups should be avoided.

(ARTIQ Python does not support calling superclass implementations in a polymorphic way – `Fragment.device_setup(self)` could be used from one concrete fragment in the whole project, but would cause the argument type to be inferred as that subclass. Only direct member function calls are special-cased to be generic on the *self* type.)

build_fragment(**args*, ***kwargs*)→ None [source]

Initialise this fragment, building up the hierarchy of subfragments, parameters and result channels.

This is where any constructor-type initialisation should take place, similar to the role `build()` has for a bare `HasEnvironment`.

While this method executes, the various `setattr_*()` functions can be used to create subfragments, parameters, and result channels.

Parameters:
- **args** – Any extra arguments that were passed to the `HasEnvironment` constructor.
- **kwargs** – Any extra keyword arguments that were passed to the `HasEnvironment` constructor.

setattr_fragment(*name: str*, *fragment_class: type[Fragment]*, **args*, ***kwargs*)→ Fragment [source]

Create a subfragment of the given name and type.

Can only be called during `build_fragment()`.

Parameters:
- **name** – The fragment name; part of the fragment path. Must be a valid Python identifier; the fragment will be accessible as `self.<name>`.

- **fragment_class** – The type of the subfragment to instantiate.
- **args** – Any extra arguments to forward to the subfragment `build_fragment()` call.
- **kwargs** – Any extra keyword arguments to forward to the subfragment `build_fragment()` call.

Returns: The newly created fragment instance.

setattr_param(*name: str*, *param_class: type[ParamBase]*, *description: str*, **args*, ***kwargs*) → ParamHandle [source]

Create a parameter of the given name and type.

Can only be called during `build_fragment()`.

Parameters:
- **name** – The parameter name, to be part of its FQN. Must be a valid Python identifier; the parameter handle will be accessible as `self.<name>`.
- **param_class** – The type of parameter to instantiate.
- **description** – The human-readable parameter name.
- **args** – Any extra arguments to pass to the `param_class` constructor.
- **kwargs** – Any extra keyword arguments to pass to the the `param_class` constructor.

Returns: The newly created parameter handle.

setattr_param_like(*name: str*, *original_owner: Fragment*, *original_name: str | None = None*, ***kwargs*) → ParamHandle [source]

Create a new parameter using an existing parameter as a template.

The newly created parameter will inherent its type, and all the metadata that is not overridden by the optional keyword arguments, from the template parameter.

This is often combined with `bind_param()` to rebind parameters from one or more subfragments; see also `setattr_param_rebind()`, which combines the two.

Can only be called during `build_fragment()`.

Parameters:
- **name** – The new parameter's name, to be part of its FQN. Must be a valid Python identifier; the parameter handle will be accessible as `self.<name>`.
- **original_owner** – The fragment owning the parameter to use as a template.
- **original_name** – The name of the parameter to use as a template (i.e. `<original_owner>.<original_name>`). If `None`, defaults to `name`.
- **kwargs** – Any attributes to override in the template parameter metadata.

Returns: The newly created parameter handle.

setattr_param_rebind(*name: str*, *original_owner: Fragment*, *original_name: str | None = None*, ***kwargs*) → ParamHandle [source]

Convenience function combining `setattr_param_like()` and `bind_param()` to override a subfragment parmeter.

The most common use case for this is to specialise the operation of a generic subfragment. For example, there might be a fragment `Fluoresce` that drives a cycling transition in an ion with parameters for intensity and detuning. Higher-level fragments for Doppler cooling, readout, etc. might then use `Fluoresce`, rebinding its intensity and detuning parameters to values and defaults appropriate for those particular tasks.

Can only be called during `build_fragment()`.

Parameters:
- **name** – The parameter name, to be part of its FQN. Must be a valid Python identifier; the parameter handle will be

accessible as `self.<name>`.
- **original_owner** – The fragment owning the parameter to rebind.
- **original_name** – The name of the original parameter (i.e. `<original_owner>.<original_name>`). If `None`, defaults to `name`.
- **kwargs** – Any attributes to override in the parameter metadata, which defaults to that of the original parameter.

Returns: The newly created parameter handle.

setattr_result(*name: str*, *channel_class: type[~ndscan.experiment.result_channels.ResultChannel] = <class 'ndscan.experiment.result_channels.FloatChannel'>*, **args*, ***kwargs*) → ResultChannel [source]

Create a result channel of the given name and type.

Can only be called during `build_fragment()`.

Parameters:
- **name** – The result channel name, to be part of its full path. Must be a valid Python identifier. The channel instance will be accessible as `self.<name>`.
- **channel_class** – The type of result channel to instantiate.
- **args** – Any extra arguments to pass to the `channel_class` constructor.
- **kwargs** – Any extra keyword arguments to pass to the the `channel_class` constructor.

Returns: The newly created result channel instance.

override_param(*param_name: str*, *initial_value: Any = None*) → tuple[Any, ParamStore] [source]

Override the parameter with the given name and set it to the provided value.

See `bind_param()`, which also overrides the parameter, but sets it to follow another parameter instead.

Parameters:
- **param_name** – The name of the parameter.
- **initial_value** – The initial value for the parameter. If `None`, the default from the parameter schema is used.

Returns: A tuple `(param, store)` of the parameter metadata and the newly created `ParamStore` instance that the parameter handles are now bound to.

bind_param(*param_name: str*, *source: ParamHandle*) → Any [source]

Override the fragment parameter with the given name such that its value follows that of another parameter.

The most common use case for this is to specialise the operation of a generic subfragment. For example, there might be a fragment `Fluoresce` that drives a cycling transition in an ion with parameters for intensity and detuning. Higher-level fragments for Doppler cooling, readout, etc. might then use `Fluoresce`, binding its intensity and detuning parameters to values and defaults appropriate for those particular tasks.

See `override_param()`, which sets the parameter to a fixed value/store.

Parameters:
- **param_name** – The name of the parameter to be bound (i.e. `self.<param_name>`). Must be a free parameter of this fragment (not already bound or overridden).
- **source** – The parameter to bind to. Must be a free parameter of its respective owner.

detach_fragment(*fragment: Fragment*) → None [source]

Detach a subfragment from the execution machinery, causing its setup and cleanup methods not to be invoked and its result channels not to be collected.

Its parameters will still be available in the global tree as usual, but the the actual execution can be customised this way, e.g. for the implementation of subscans.

Parameters: **fragment** – The fragment to detach; must be a direct subfragment of this fragment.

init_params(*overrides: dict[str, list[tuple[str, ParamStore]]] = {}*) → None [source]

Initialise free parameters of this fragment and all its subfragments.

If, for a given parameter, a relevant override is given, the specified ParamStore is used. Otherwise, the default value is evaluated and a new store pointing to it created.

This method should be called before any of the fragment's user-defined functions are used (but after the constructor -> `build()` -> :meth`build_fragment()` has completed). Most likely, the top-level fragment will be called from an `ndscan.experiment.entry_point`, which already take care of this. In cases where fragments are used in a different context, for example from a standalone `EnvExperiment`, this method must be called manually.

Parameters: **overrides** – A dictionary mapping parameter FQNs to lists of overrides. Each override is specified as a tuple *(pathspec, store)* of a path spec and the store to use for parameters the path of which matches the spec.

recompute_param_defaults() → None [source]

Recompute default values of the parameters of this fragment and all its subfragments.

For parameters where the default value was previously used, the expression is evaluated again – thus for instance fetching new dataset values –, and assigned to the existing parameter store. An informative message is logged if the value changed, such that changes remain traceable after the fact (e.g. when an experiment is resumed after a calibration interruption).

make_namespaced_identifier(*name: str*) → str [source]

Mangle passed name and path to this fragment into a string, such that calls from different fragments give different results for the same name.

This can, for instance, be useful when naming DMA sequences, or interacting with other kinds of global registries, where multiple fragment instances should not conflict with each other.

The returned string will consist of characters that are valid Python identifiers and slashes.

get_always_shown_params() → list[ParamHandle] [source]

Return handles of parameters to always show in user interfaces when this fragment is the root of the fragment tree (vs. other parameters for which overrides need to be explicitly added).

This can be overridden in fragment implementations to customise the user interface; by default, all free parameters are shown in the order they were created in `build_fragment()`.

Example:

```
class MyFragment(Fragment):
    def build_fragment(self):
        self.setattr_fragment("child", MyChildFragment)
        self.setattr_param("foo", ...)
        self.setattr_param("bar", ...)
        self.setattr_param("baz", ...)

    def get_always_shown_params(self):
        shown = super().get_always_shown_params()

        # Don't show self.bar.
```

```
        shown.remove(self.bar)

        # Always show an important parameter from a
        # child fragment.
        shown.add(self.child.very_important_param)

        return shown
```

class **ndscan.experiment.fragment.ExpFragment**(*managers_or_parent*, **args*, ***kwargs*) [source]

Fragment that supports the notion of being run to produce results.

prepare()→ None [source]

Prepare this instance for execution (see `artiq.language.environment.Experiment.prepare`).

This is invoked only once per (sub)scan, after `Fragment.build_fragment()` but before `host_setup()`. At this point, parameters, datasets and devices can be accessed, but devices must not yet be.

For top-level scans, this can (and will) be executed in the *prepare* scheduler pipeline stage.

Unless running in the *prepare* pipeline state is absolutely necessary for runtime performance, lazily running the requisite initialisation code in `host_setup()` is usually preferable, as this naturally composes across the ndscan fragment tree.

run_once()→ None [source]

Execute the experiment described by the fragment once with the current parameters, producing one set of results (if any).

get_default_analyses()→ Iterable[DefaultAnalysis] [source]

Return list of `DefaultAnalysis` instances describing analyses (fits, etc.) for this fragment.

Analyses are only run if the fragment is scanned along the axes required for them to apply.

This is a class method in spirit, and might become one in the future.

class **ndscan.experiment.fragment.AggregateExpFragment**(*managers_or_parent*, **args*, ***kwargs*) [source]

Combines multiple `ExpFragment`s and callables by executing them one after each other each time `run_once()` is called.

To use, derive from the class and, in the subclass `build_fragment()` method forward to the parent implementation after constructing all the relevant fragments:

```
class FooBarAggregate(AggregateExpFragment):
    def build_fragment(self):
        self.setattr_fragment("foo", FooFragment)
        self.setattr_fragment("bar", BarFragment)

        # Any number of customisations can be made as usual,
        # e.g. to provide a convenient parameter to scan the
        # fragments in lockstep:
        self.setattr_param_rebind("freq", self.foo)
        self.bar.bind_param("freq", self.freq)

        _, bar_store = self.bar.override_param("param")

        @kernel
        def forward() -> None:
            # Set the value of one of bar's parameters based on
            # one of foo's results
            bar_store.set_value(self.foo.result.get_last())

        # Let AggregateExpFragment's default implementations
        # take care of the rest, e.g. have self.run_once()
        # call self.foo.run_once(), then forward() and, finally,
        # self.bar.run_once()
        super().build_fragment([self.foo, forward, self.bar])
```

```
ScanFooBarAggregate = make_fragment_scan_exp(FooBarAggregate)
```

Each aggregated experiment should be independent, i.e. do its own setup and clean up.

build_fragment(*operands: list[Callable[[], None] | ExpFragment]*) → None [source]

Parameters: **operands** – The list of objects to be aggregated. Each item may either be a "child" fragment or a callable. The operands will be run in the given order, calling `run_once` on child fragments. No special treatment is given to the `{host,device}_{setup,cleanup}()` methods, which will be executed through the recursive default implementations unless overridden by the user.

run_once() → None [source]

Execute the experiment by calling all operands.

Invokes all operands in the order they are passed to `build_fragment()`. This method can be overridden if more complex behaviour is desired.

If all operands have a `@kernel` `run_once()`, this is implemented on the core device as well to avoid costly kernel recompilations in a scan.

get_always_shown_params() → list[ParamHandle] [source]

Collect always-shown params from each child fragment, plus any parameters directly defined in this fragment as usual.

get_default_analyses() → Iterable[DefaultAnalysis] [source]

Collect default analyses from each child fragment.

The analyses are wrapped in a proxy that prepends any result channel names with the fragment path to ensure results from different analyses do not collide.

exception **ndscan.experiment.fragment.TransitoryError** [source]

Transitory error encountered while executing a fragment, which is expected to clear itself up if it is attempted again without any further changes.

Such errors are never raised by the ndscan infrastructure itself, but can be thrown from user fragment implementations in response to e.g. some temporary hardware conditions such as a momentarily insufficient level of laser power.

`ndscan.experiment.entry_point`s will attempt to handle transitory errors, e.g. by retrying execution some amount of times. If fragments are manually executed from user code, it will often be appropriate to do this as well, unless the user code base does not use transitory errors at all.

exception **ndscan.experiment.fragment.RestartKernelTransitoryError** [source]

`TransitoryError` where, as part of recovering from it, the kernel should be restarted before retrying.

This can be used for cases where remedying the error requires `Fragment.host_setup()` to be run again, such as for cases where the experiments needs to yield back to the scheduler (e.g. for an ion loss event to be remedied by a second reloading experiment).

`ndscan.experiment.parameters` module

Fragment-side parameter containers.

In practical use, these will be instantiated by calling `Fragment.setattr_param()` with the appropriate type argument (`FloatParam`, `IntParam`, `StringParam`, `BoolParam`, `EnumParam`).

exception **ndscan.experiment.parameters.InvalidDefaultError** [source]

Raised when a default value is outside the specified range of valid parameter values.

class **ndscan.experiment.parameters.ParamStore**(*identity: tuple[str, str]*, *value*) [source]

Parameters:
- **identity** – `(fqn, path_spec)` pair representing the identity of this param store, i.e. the override/default value it was created for.
- **value** – The initial value.

RpcType

The type to use for this parameter in the RPC layer (to be overridden by subclasses).

alias of `Any`

to_rpc_type(*value*) → Any [source]

For types that need to be represented differently in the RPC layer (enums), convert the value from overrides/scan generators/etc. to the type used across the RPC interface.

set_from_rpc(*value*) → None [source]

For types that need to be represented differently in the RPC layer (enums), convert the value back to the type used in the kernel.

class **ndscan.experiment.parameters.ParamHandle**(*owner: Fragment*, *name: str*, *parameter*) [source]

Each instance of this class corresponds to exactly one attribute of a fragment that can be used to access the underlying parameter store.

Parameters:
- **owner** – See `owner`.
- **name** – See `name`.
- **parameter** – The parameter initially associated with this handle (see `parameter`).

owner

The `Fragment` owning this parameter handle.

name

The name of the attribute in the owning fragment that corresponds to this object.

parameter

Points to the parameter currently associated with this handle, tracking binding of the parameter.

`ndscan.experiment.result_channels` module

Result handling building blocks.

class **ndscan.experiment.result_channels.SingleUseSink** [source]

Sink that allows only one value to be pushed (before being cleared).

push(*value: Any*) → None [source]

Record a new value.

This should never fail; neither in the sense of raising an exception, nor (for sinks that record a series of values) in the sense of failing to record the presence of a value, as code consuming results relies on one `push()` each to a set of result channels and subsequently sinks representing a single multi- dimensional data point.

class **ndscan.experiment.result_channels.LastValueSink** [source]

Sink that allows multiple values to be pushed, but retains only the last-pushed one.

push(*value: Any*)→ None [source]

Record a new value.

This should never fail; neither in the sense of raising an exception, nor (for sinks that record a series of values) in the sense of failing to record the presence of a value, as code consuming results relies on one `push()` each to a set of result channels and subsequently sinks representing a single multi- dimensional data point.

get_last()→ Any [source]

Return the last-pushed value, or `None` if none yet.

class **ndscan.experiment.result_channels.ArraySink** [source]

Sink that stores all pushed values in a list.

push(*value: Any*)→ None [source]

Record a new value.

This should never fail; neither in the sense of raising an exception, nor (for sinks that record a series of values) in the sense of failing to record the presence of a value, as code consuming results relies on one `push()` each to a set of result channels and subsequently sinks representing a single multi- dimensional data point.

get_all()→ list[Any] [source]

Return a list of all previously pushed values.

get_last()→ Any [source]

Return the last-pushed value, or `None` if none yet.

clear()→ None [source]

Clear the list of previously pushed values.

class **ndscan.experiment.result_channels.ScalarDatasetSink**(*managers_or_parent*, **args*, ***kwargs*) [source]

Sink that writes pushed results to a dataset, overwriting its previous value if any.

build(*key: str*, *broadcast: bool = True*)→ None [source]

Parameters:
- **key** – Dataset key to write the value to.
- **broadcast** – Whether to set the dataset in broadcast mode.

push(*value: Any*)→ None [source]

Record a new value.

This should never fail; neither in the sense of raising an exception, nor (for sinks that record a series of values) in the sense of failing to record the presence of a value, as code consuming results relies on one `push()` each to a set of result channels and subsequently sinks representing a single multi- dimensional data point.

get_last()→ Any [source]

Return the last pushed value, or `None` if none yet.

class **ndscan.experiment.result_channels.ResultChannel**(*path: str*, *description: str =*, *display_hints: dict[str, ~typing.Any] | None = None*, *save_by_default: bool = True*) [source]

Parameters:

- **path** – The path to the channel in the fragment tree (e.g. `"readout/p"`).
- **description** – A human-readable name of the channel. If non-empty, will be preferred to the path to e.g. display in plot axis labels.
- **display_hints** –
  A dictionary of additional settings that can be used to indicate how to best display results to the user (see above):

Key	Argument	Description
`coordinate_type`	String describing the coordinate type.	For numeric channels, describes the coordinate system for the resulting values, which can be used to select a more appropriate visualisation than the default, which corresponds to straightforward linear coordinates (optionally bounded if `min` / `max` are set). Currently implemented: `cyclic`, where the values are cyclical between `min` and `max` (e.g. a phase between 0 and 2π).
`error_bar_for`	Path of the linked result channel	Indicates that this (numeric) result channel should be used to determine the size of the error bars for the given other channel.
`priority`	Integer	Specifies a sort order between result channels, used e.g. to control the way various axes are laid out. Channels are sorted from highest to lowest priority (default: 0). Channels with negative priorities are not displayed by default unless explicitly enabled.
`share_axis_with`	Path of the linked result channel	Indicates that this result channel should be drawn on the same plot axis as the given other channel.
`share_pane_with`	Path of the linked result channel	Indicates that this result channel should be drawn on the same plot pane as the given other channel (but e.g. on its own y axis). This restores the behaviour of previous `ndscan` versions, where all axes used to be shown in a single plot pane.

class **ndscan.experiment.result_channels.NumericChannel**(*path: str*, *description: str =*, *display_hints: dict[str, ~typing.Any] | None = None*, *min=None*, *max=None*, *unit: str =*, *scale=None*)
[source]

Base class for `ResultChannel`s of numerical results, with scale/unit semantics and optional range limits.

Parameters:

- **min** – Optionally, a lower limit that is not exceeded by data points (can be used e.g. by plotting code to determine sensible value ranges to show).
- **max** – Optionally, an upper limit that is not exceeded by data points (can be used e.g. by plotting code to determine sensible value ranges to show).
- **unit** – Name of the unit the results are given in (e.g. `"ms"`, `"kHz"`).

- **scale** – Unit scaling. If `None`, the default scaling as per ARTIQ's unit handling machinery (`artiq.language.units`) is used.

get_last() [source]

Returns the last value pushed to this result channel.

This method is a workaround for limitations of ARTIQ python, which make it impractical to extract values from the sinks without going through RPCs.

class **ndscan.experiment.result_channels.FloatChannel**(*path: str*, *description: str =*, *display_hints: dict[str, ~typing.Any] | None = None*, *min=None*, *max=None*, *unit: str =*, *scale=None*) [source]

`NumericChannel` that accepts floating-point results.

class **ndscan.experiment.result_channels.IntChannel**(*path: str*, *description: str =*, *display_hints: dict[str, ~typing.Any] | None = None*, *min=None*, *max=None*, *unit: str =*, *scale=None*) [source]

`NumericChannel` that accepts integer results.

class **ndscan.experiment.result_channels.OpaqueChannel**(*path: str*, *description: str =*, *display_hints: dict[str, ~typing.Any] | None = None*, *save_by_default: bool = True*) [source]

`ResultChannel` that stores arbitrary data, with ndscan making no attempts to further interpret or display it.

As such, opaque channels can be used to store any ancillary data for scan points, which can later be used in custom analysis code (whether as part of a default analysis that runs as part of the experiment code, or when manually analysing the experimental data later).

Any values pushed are just passed through to the ARTIQ dataset layer; it is up to the user to choose something compatibile with HDF5 and PYON.

Scans

`ndscan.experiment.scan_generator` module

class **ndscan.experiment.scan_generator.ScanGenerator** [source]

Generates points along a single scan axis to be visited.

class **ndscan.experiment.scan_generator.RefiningGenerator**(*lower*, *upper*, *randomise_order*) [source]

Generates progressively finer grid by halving distance between points each level.

class **ndscan.experiment.scan_generator.LinearGenerator**(*start*, *stop*, *num_points*, *randomise_order*) [source]

Generates equally spaced points between two endpoints.

class **ndscan.experiment.scan_generator.ListGenerator**(*values*, *randomise_order*) [source]

Generates points by reading from an explicitly specified list.

class **ndscan.experiment.scan_generator.CentreSpanRefiningGenerator**(*centre*, *half_span*, *randomise_order*, *limit_lower=-inf*, *limit_upper=inf*) [source]

Generates progressively finer grid in span around centre by halving distance between points each level. Scan is always centred on the given centre, even if the span exceeds the limits. :param limit_lower: Optional lower limit (inclusive) to the range of generated

points. Useful for representing scans on parameters the range of which is limited (e.g. to be non-negative).

Parameters: **limit_upper** – See *limit_lower*.

`ndscan.experiment.scan_runner` module

Generic scanning loop.

While `scan_generator` describes a scan to be run in the abstract, this module contains the implementation to actually execute one within an ARTIQ experiment. This will likely be used by end users via `FragmentScanExperiment` or subscans.

class **ndscan.experiment.scan_runner.ScanAxis**(*param_schema: dict[str, Any]*, *path: str*, *param_store: ParamStore*) [source]

Describes a single axis that is being scanned.

Apart from the metadata, this also includes the necessary information to execute the scan at runtime; i.e. the `ParamStore` to modify in order to set the parameter.

class **ndscan.experiment.scan_runner.ScanSpec**(*axes: list[ScanAxis]*, *generators: list[ScanGenerator]*, *options: ScanOptions*) [source]

Describes a single scan.

axes: *list[ScanAxis]*

The list of parameters that are scanned.

generators: *list[ScanGenerator]*

Generators that give the points for each of the specified axes.

options: *ScanOptions*

Applicable `ScanOptions`.

class **ndscan.experiment.scan_runner.ScanRunner**(*managers_or_parent*, **args*, ***kwargs*) [source]

Runs the actual loop that executes an `ExpFragment` for a specified list of scan axes (on either the host or core device, as appropriate).

build(*max_rtio_underflow_retries: int = 3*, *max_transitory_error_retries: int = 10*, *skip_on_persistent_transitory_error: bool = False*) [source]

Parameters:

- **max_rtio_underflow_retries** – Number of RTIOUnderflows to tolerate per scan point (by simply trying again) before giving up. Three is a pretty arbitrary default – we don't want to block forever in case the experiment is faulty, but also want to tolerate ~1% underflow chance for experiments where tight timing is critical.
- **max_transitory_error_retries** – Number of transitory errors to tolerate per scan point (by simply trying again) before giving up.
- **skip_on_persistent_transitory_error** – By default, transitory errors above the configured limit are raised for the calling code to handle (possibly terminating the experiment). If `True`, points with too many transitory errors will be skipped instead after logging an error. Consequences for overall system robustness should be considered before using this in automated code.

run(*fragment: ExpFragment*, *spec: ScanSpec*, *axis_sinks: list[ResultSink]*) → None [source]

Run a scan of the given fragment, with axes as specified.

Integrates with the ARTIQ scheduler to pause/terminate execution as requested.

Parameters:

- **fragment** – The fragment to iterate.
- **options** – The options for the scan generator.

- **axis_sinks** – A list of `ResultSink` instances to push the coordinates for each scan point to, matching `scan.axes`.

acquire()→ bool [source]

Returns: `true` if scan is complete, `false` if the scan has been interrupted and `acquire()` should be called again to complete it.

ndscan.experiment.scan_runner.match_default_analysis(*analysis: DefaultAnalysis*, *axes: Iterable[ScanAxis]*)→ bool [source]

Return whether the given default analysis can be executed for the given scan axes.

The implementation is currently a bit more convoluted than necessary, as we want to catch cases where the parameter specified by the analysis is scanned indirectly through overrides. (TODO: Do we really, though? This matches the behaviour prior to the refactoring towards exposing a set of required axis handles from DefaultAnalysis, but we should revisit this.)

ndscan.experiment.scan_runner.filter_default_analyses(*fragment: ExpFragment*, *axes: Iterable[ScanAxis]*)→ list[DefaultAnalysis] [source]

Return the default analyses of the given fragment that can be executed for the given scan spec.

See `match_default_analysis()`.

ndscan.experiment.scan_runner.describe_scan(*spec: ScanSpec*, *fragment: ExpFragment*, *short_result_names: dict[ResultChannel, str]*)→ dict[str, Any] [source]

Return metadata for the given spec in stringly typed dictionary form.

Parameters:
- **spec** – `ScanSpec` describing the scan.
- **fragment** – Fragment being scanned.
- **short_result_names** – Map from result channel objects to shortened names.

ndscan.experiment.scan_runner.describe_analyses(*analyses: Iterable[DefaultAnalysis]*, *context: AnnotationContext*)→ dict[str, Any] [source]

Return metadata for the given analyses in stringly typed dictionary form.

Parameters:
- **analyses** – The `DefaultAnalysis` objects to describe (already filtered to those that apply to the scan, and thus are describable by the context).
- **context** – Used to resolve any references to scanned parameters/results channels/analysis results.

Returns: The analysis metadata (`annotations` / `online_analyses`), with all references to fragment tree objects resolved, and ready for JSON/… serialisation.

`ndscan.experiment.subscan` module

Implements subscans, that is, the ability for an `ExpFragment` to scan another child fragment as part of its execution.

ndscan.experiment.subscan.setattr_subscan(*owner: Fragment*, *scan_name: str*, *fragment: ExpFragment*, *axis_params: list[tuple[Fragment, str]]*, *save_results_by_default: bool = True*, *expose_analysis_results: bool = True*)→ Subscan [source]

Set up a scan for the given subfragment.

Result channels are set up in the owning fragment to expose the scan data, such that scan results can be inspected after the fact.

This is the legacy subscan interface, and is geared primarily towards executing the scan loop on the host by calling `Subscan.run()` on the returned handle, which takes care of setup/results management/etc. all at once. To be able to execute scans on-kernel, `SubscanExpFragment` is preferred, as it directly integrates the lifecycle

management with the usual setup/cleanup methods, which is more convenient in that case.

Parameters:
- **owner** – The fragment to add the subscan to.
- **scan_name** – Name of the scan; appears in result channel names, and the `Subscan` instance will be available as `owner.<scan_name>`.
- **fragment** – The runnable fragment to iterate over in the scan. Must be a subfragment of `owner`.
- **axis_params** – List of *(fragment, param_name)* tuples defining the axes to be scanned. It is possible to specify more axes than are actually used; they will be overridden and set to their default values.
- **save_results_by_default** – Passed on to all derived result channels.
- **expose_analysis_results** – Whether to add result channels to `owner` that contain the results of default analyses set for the fragment. Note that for this, all results must be known when this function is called (that is, all `axis_params` should actually be scanned, and the analysis must not fail to produce results).

Returns: A `Subscan` instance to use to actually execute the scan.

---

class **ndscan.experiment.subscan.Subscan**(*runner: ScanRunner, fragment: ExpFragment, possible_axes: dict[ParamHandle, ScanAxis], schema_channel: SubscanChannel, coordinate_channels: list[ResultChannel], child_result_sinks: dict[ResultChannel, ArraySink], aggregate_result_channels: dict[ResultChannel, ResultChannel], short_child_channel_names: dict[str, ResultChannel], analyses: list[DefaultAnalysis], parent_analysis_result_channels: dict[str, ResultChannel]*) [source]

Handle returned by `setattr_subscan()`, allowing the subscan to actually be executed.

run(*axis_generators: list[tuple[ParamHandle, ScanGenerator]], options: ScanOptions = ScanOptions(num_repeats=1, num_repeats_per_point=1, randomise_order_globally=False, seed=90782297), execute_default_analyses: bool = True*) → tuple[dict[ParamHandle, list], dict[ResultChannel, list]] [source]

Run the subscan with the given axis iteration specifications, and return the data point coordinates/result channel values.

Parameters:
- **axis_generators** – The list of scan axes (dimensions). Each element is a tuple of parameter to scan (handle must have been passed to `setattr_subscan()` to set up), and the `ScanGenerator` to use to generate the points.
- **options** – `ScanOptions` to control scan execution.
- **execute_default_analyses** – Whether to run any default analyses associated with the subfragment after the scan is complete, even if they are not exposed as owning fragment channels.

Returns: A tuple `(coordinates, values, analysis_results)`, each a dictionary mapping parameter handles, result channels and analysis channel names to lists of their values.

---

class **ndscan.experiment.subscan.SubscanExpFragment**(*managers_or_parent, *args, **kwargs*) [source]

An `ExpFragment` that scans another `ExpFragment` when it executes (“subscan”).

Compared to the legacy way of creating subscans, `setattr_subscan()`, this seamlessly supports the execution of `@kernel` subscans: not only can the scanned fragment be run on the core device (which the legacy interface supported as well), but the `run_once()` method driving the scan itself can also be `@kernel`. This means that `SubscanExpFragment` can be used as part of bigger on-device experiments, and that frequent recompilation overhead for repeated subscans can be avoided.

The API of this fragment supports use through composition, which is the natural and more flexible way (compared to inheritance). However, when using such a fragment as part of a larger code base, be aware of the general restrictions of the ARTIQ

Python compiler, in particular the fact that all instances of a class must share the same type (including attributes, etc.). For this reason, you might want to create a separate subtype of this class for each use, such that multiple pieces of client code remain composable (can be combined into yet another bigger on-kernel program). One way to achieve this is by just creating an "empty" subclass:

```
class Foo(ExpFragment):
    "The fragment to be scanned."
    def build_fragment(self) -> None:
        self.setattr_param("param_a", FloatParam, "a value", default=0.0)
        # […]

    @kernel
    def run_once(self):
        # […]

class FooSubscan(SubscanExpFragment):
    pass

class Parent(ExpFragment):
    def build_fragment(self) -> None:
        self.setattr_fragment("foo", Foo)
        self.setattr_fragment("scan", FooSubscan, self, "foo",
            [(self.foo, "param_a")])
        self.setattr_param("num_scan_points",
                           IntParam,
                           "Number of scan points",
                           default=21,
                           min=2)

    @rpc(flags={"async"})
    def configure_scan(self):
        if self.num_scan_points.changed_after_use():
            self.scan.configure([(self.foo.param_a,
                LinearGenerator(0.0, 0.1, self.num_scan_points.use()))])

    def host_setup(self):
        # Run at least once before kernel starts such that all the fields
        # are initialised (required for the ARTIQ compiler).
        self.configure_scan()
        super().host_setup()

    @kernel
    def device_setup(self):
        # Update scan if num_scan_points was changed (can be left out if
        # there are no scannable parameters influencing the scan settings).
        self.configure_scan()
        self.device_setup_subfragments()

    @kernel
    def run_once(self):
        # Execute the subscan (and anything else that the fragment might
        # need to do).
        self.scan.run_once()
```

Another way is to just make the `ExpFragment` performing the subscan a subclass of `SubscanExpFragment`:

```
class Parent(SubscanExpFragment):
    def build_fragment(self) -> None:
        self.setattr_fragment("foo", Foo)
        super().build_fragment(self, "foo", [(self.foo, "param_a")])
        self.setattr_param("num_scan_points",
                           IntParam,
                           "Number of scan points",
                           default=21,
                           min=2)

    # configure_scan(), host_setup() and device_setup() as above.
```

build_fragment(*scanned_fragment_parent: Fragment, scanned_fragment: ExpFragment | str, axis_params: list[tuple[Fragment, str]], save_results_by_default: bool = True, expose_analysis_results: bool = True*) → None [source]

Parameters:

- **scanned_fragment_parent** – The fragment that owns the scanned fragment.
- **scanned_fragment** – The fragment to scan. Can either be passed as a string (the name of the fragment in the parent) or directly as the `ExpFragment` reference.

- **axis_params** – List of *(fragment, param_name)* tuples defining the axes to be scanned.
- **save_results_by_default** – Passed on to all derived result channels.
- **expose_analysis_results** – Whether to add result channels to this fragment that contain the results of default analyses set for the fragment. Note that for this to work, all results must be known when this function is called (that is, all `axis_params` should actually be scanned, and any analyses must not fail to produce results).

configure(*axis_generators: list[tuple[ParamHandle, ScanGenerator]], options: ScanOptions = ScanOptions(num_repeats=1, num_repeats_per_point=1, randomise_order_globally=False, seed=3433425982)*) → None [source]

Configure point generators for each scan axis, and scan options.

This only needs to be called once (but can be called multiple times to change settings between `run_once()` invocations, e.g. from a parent fragment `{host, device}_setup()`).

For on-core-device scans, this has to be called at least once before the kernel is first entered (e.g. from `host_setup()`) such that the types of all the fields can be known.

Parameters:
- **axis_generators** – The list of scan axes (dimensions). Each element is a tuple of parameter to scan (must correspond to one of the axes specified in the constructor; see `build_fragment()`), and the `ScanGenerator` to use to generate the points.
- **options** – `ScanOptions` to control scan execution.

run_once() → None [source]

Execute the subscan as previously configured.

This has the usual semantics of a fragment `run_once()` method, i.e. calling it will acquire one set of results for the fragment (here, a complete scan) and write them to the result channels. If the scanned fragment has an `@kernel` `run_once()` method, this will automatically be made a `@kernel` method as well.

`ndscan.experiment.annotations` module

Annotations are a way for analyses (`ndscan.experiment.default_analysis`) to specify additional information to be displayed to the user along with the scan data, for instance a fit curve or a line indicating a centre frequency or oscillation period derived from the analysis.

Conceptually, annotations express hints about a suitable user interface for exploration of experimental data, rather than a mechanism for data storage or exchange. To make analysis results programmatically accessible to other parts fo a complex experiment, analysis result channels (see `DefaultAnalysis.get_analysis_results()`) should be used instead.

class **ndscan.experiment.annotations.Annotation**(*kind: str, coordinates: dict | None = None, parameters: dict | None = None, data: dict | None = None*) [source]

An annotation to be displayed alongside scan result data, recording derived quantities (e.g. a fit minimizer).

See `curve()`, `curve_1d()`, `computed_curve()`, `axis_location()`.

ndscan.experiment.annotations.curve_1d(*x_axis: ParamHandle, x_values: list[float] | ndarray | AnnotationValueRef, y_axis: ResultChannel, y_values: list[float] | ndarray | AnnotationValueRef*) → Annotation [source]

Create a curve annotation from explicit lists of x and y coordinates.

This will typically be shown as a connected line in the plot applet. See `curve()` for a generic variant covering multiple dimensions (though curve annotations are currently only displayed for one-dimensional scans in the applet).

If the curve data comes from a functional relationship matching one of the predefined fit types (`ndscan.util.FIT_OBJECTS`), prefer `computed_curve_1d()` as this allows for unlimited resolution (also if the user looks at a range outside that corresponding to the source scan) and is more efficient to store.

Parameters:
- **x_axis** – The parameter corresponding to the x axis of the curve.
- **x_values** – A list of x coordinates for the curve points.
- **y_axis** – The result channel corresponding to the y axis of the curve.
- **y_values** – A list of y coordinates for the curve points.

Returns: The `Annotation` object describing the curve.

---

ndscan.experiment.annotations.curve(*coordinates: dict[ParamHandle | ResultChannel, list[float] | ndarray]*) → Annotation [source]

Create a curve annotation from a dictionary of coordinate lists.

This will typically be shown as a connected line in the plot applet. See `curve_1d()` for an alternative signature that is slightly more explicit/self-documenting for one-dimensional scans (the only type of curve supported in the plot applet at this point).

If the curve data comes from a functional relationship matching one of the predefined fit types (`ndscan.util.FIT_OBJECTS`), prefer `computed_curve()`, as this allows for unlimited resolution (also if the user looks at a range outside that corresponding to the source scan) and is more efficient to store.

Parameters: **coordinates** – A dictionary mapping, for each dimension, the axis in question (`ParamHandle` / `ResultChannel`) to a list of coordinates for each curve point. Each list must have the same length.

Returns: The `Annotation` object describing the curve.

---

ndscan.experiment.annotations.computed_curve(*function_name: str, parameters: dict[str, Any | AnnotationValueRef], associated_channels: list | None = None*) → Annotation [source]

Create a curve annotation that is computed from a well-known fit function (`ndscan.util.FIT_OBJECTS`).

This will typically be shown as a connected line in the plot applet. See `curve()` / `curve_1d()` for a variant defined by a discrete list of points instead of the evaluation of a function.

Parameters:
- **function_name** – The name of the function to use, matching the keys in `ndscan.util.FIT_OBJECTS`.
- **parameters** – The fixed parameters to evaluate the function with at each point, given as a dictionary (see `oitg.fitting.FitBase.parameter_names` for the expected keys).
- **associated_channels** – If given, the curve will be shown on the same axis/plot/etc. as the given result channels (possibly more than once). Prefer explicitly specifying this to avoid unexpected behaviour if e.g. additional result channels with different logical meanings (units, etc.) are added to the experiment later.

Returns: The `Annotation` object describing the curve.

---

ndscan.experiment.annotations.axis_location(*axis: ParamHandle | ResultChannel, position: Any | AnnotationValueRef, position_error: float | AnnotationValueRef | None = None, associated_channels: list | None = None*) → Annotation [source]

Create an annotation marking a specific location on the given axis.

This will typically be shown as a vertical/horizontal line in the plot applet (though currently only vertical lines for x axis positions are implemented on the applet side).

Parameters:
- **axis** – The parameter or result channel the location corresponds to.
- **position** – The location to mark, given in the same units as the axis or parameter value. Typically a numerical value, though once scans over non-numerical axes are implemented, this could also be one instance of a categorical value.
- **position_error** – Optionally, the uncertainty ("error bar") associated with the given position.
- **associated_channels** – If given, the curve will be shown on the same axis/plot/etc. as the given result channels (possibly more than once). Prefer explicitly specifying this to avoid unexpected behaviour if e.g. additional result channels with different logical meanings (units, etc.) are added to the experiment later.

Returns: The `Annotation` object describing the curve.

`ndscan.experiment.default_analysis` module

Interfaces and declarations for analyses.

Conceptually, analyses are attached to a fragment, and produce results "the next level up" – that is, they condense all the points from a scan over a particular choice of parameters into a few derived results.

Two modalities are supported:

- Declarative fits of a number of pre-defined functions, to be executed locally by the user interface displaying the result data, and updated as data continues to accumulate ("online analysis").
- A separate analysis step executed at the end, after a scan has been completed. This is the equivalent of ARTIQ's `EnvExperiment.analyze()`, and is executed within the master worker process ("execute an analysis", "analysis results").

Both can produce `Annotation`s; particular values or plot locations highlighted in the user interface.

class **ndscan.experiment.default_analysis.Annotation**(*kind: str, coordinates: dict | None = None, parameters: dict | None = None, data: dict | None = None*) [source]

An annotation to be displayed alongside scan result data, recording derived quantities (e.g. a fit minimizer).

See `curve()`, `curve_1d()`, `computed_curve()`, `axis_location()`.

class **ndscan.experiment.default_analysis.DefaultAnalysis** [source]

Analysis functionality associated with an *ExpFragment* to be executed when that fragment is scanned in a particular way.

required_axes()→ set[ParamHandle] [source]

Return the scan axes necessary for the analysis to apply, in form of the parameter handles.

describe_online_analyses(*context: AnnotationContext*)→ tuple[list[dict[str, Any]], dict[str, dict[str, Any]]] [source]

Serialise information about online analyses to stringly typed metadata.

Parameters: **context** – The `AnnotationContext` to use to resolve references to fragment tree objects in user-specified data.

Returns: A tuple of string dictionary representations for annotations and online analyses (with all the fragment tree references resolved).

get_analysis_results()→ dict[str, ResultChannel] [source]

Return `ResultChannel`s for the results produced by the analysis, as a dictionary indexed by name.

execute(*axis_data: dict[tuple[str, str], list], result_data: dict[ResultChannel, list], context: AnnotationContext*) → list[dict[str, Any]] [source]

Exceute analysis and serialise information about resulting annotations to stringly typed metadata.

Parameters: **context** – The AnnotationContext to use to describe the coordinate axes/ result channels in the resulting metadata.

Returns: A list of string dictionary representations for the resulting annotations, if any.

---

class **ndscan.experiment.default_analysis.CustomAnalysis**(*required_axes: Iterable[ParamHandle], analyze_fn: Callable[[dict[ParamHandle, list], dict[ResultChannel, list], dict[str, ResultChannel]], list[Annotation] | None], analysis_results: Iterable[ResultChannel] = []*) [source]

`DefaultAnalysis` that executes a user-defined analysis function in the `execute()` step.

No analysis is run online.

Parameters:

- **required_axes** – List/set/… of parameters that are required as inputs for the analysis to run (given by their `ParamHandle`s). The order of elements is inconsequential.
- **analyze_fn** –
  The function to invoke in the analysis step. It is passed three dictionaries:
  1. a map from parameter handles to lists of the respective values for eachscan point,
  2. a map from result channels to lists of results for each scan point,
  3. channels for each of the optional analysis results specified in `analysis_results`, given as a dictionary indexed by channel name.

  For backwards-compatibility, the third parameter can be omitted. Optionally, a list of annotations to broadcast can be returned.
- **analysis_results** – Optionally, a number of result channels for analysis results. They are later passed to `analyze_fn`.

---

class **ndscan.experiment.default_analysis.OnlineFit**(*fit_type: str, data: dict[str, ParamHandle | ResultChannel], annotations: dict[str, dict[str, Any]] | None = None, analysis_identifier: str = None, constants: dict[str, Any] | None = None, initial_values: dict[str, Any] | None = None, save_fit_results: bool | None = None*) [source]

Describes an automatically executed fit for a given combination of scan axes and result channels.

Parameters:

- **fit_type** – Fitting procedure name, per `FIT_OBJECTS`.
- **data** – Maps fit data axis names (`"x"`, `"y"`) to parameter handles or result channels that supply the respective data.
- **annotations** – Any points of interest to highlight in the fit results, given in the form of a dictionary mapping (arbitrary) identifiers to dictionaries mapping coordinate names to fit result names. If `None`, `DEFAULT_FIT_ANNOTATIONS` will be queried.
- **analysis_identifier** – Optional explicit name to use for online analysis. Defaults to `fit_<fit_type>`, but can be set explicitly to allow more than one fit of a given type at a time.
- **constants** – Specifies parameters to be held constant during the fit. This is a dictionary mapping fit parameter names to the respective constant values, forwarded to `oitg.fitting.FitBase.FitBase.fit()`.
- **initial_values** – Specifies initial values for the fit parameters. This is a dictionary mapping fit parameter names to the respective values, forwarded to `oitg.fitting.FitBase.FitBase.fit()`.

class **ndscan.experiment.default_analysis.ResultPrefixAnalysisWrapper**(*wrapped: DefaultAnalysis*, *prefix: str*) [source]

Wraps another default analysis, prepending the given string to the name of each analysis result.

This can be used to disambiguate potential conflicts between result names when programmatically collecting analyses from multiple sources.

required_axes()→ set[ParamHandle] [source]

Return the scan axes necessary for the analysis to apply, in form of the parameter handles.

describe_online_analyses(*context: AnnotationContext*)→ tuple[list[dict[str, Any]], dict[str, dict[str, Any]]] [source]

Serialise information about online analyses to stringly typed metadata.

Parameters: **context** – The `AnnotationContext` to use to resolve references to fragment tree objects in user-specified data.

Returns: A tuple of string dictionary representations for annotations and online analyses (with all the fragment tree references resolved).

get_analysis_results()→ dict[str, ResultChannel] [source]

Return `ResultChannel`s for the results produced by the analysis, as a dictionary indexed by name.

execute(*axis_data: dict[tuple[str, str], list]*, *result_data: dict[ResultChannel, list]*, *context: AnnotationContext*)→ list[dict[str, Any]] [source]

Exceute analysis and serialise information about resulting annotations to stringly typed metadata.

Parameters: **context** – The AnnotationContext to use to describe the coordinate axes/ result channels in the resulting metadata.

Returns: A list of string dictionary representations for the resulting annotations, if any.

Experiment entry points

`ndscan.experiment.entry_point` module

Top-level functions for launching `ExpFragment`s and their scans from the rest of the ARTIQ `HasEnvironment` universe.

The two main entry points into the `ExpFragment` universe are

- scans (with axes and overrides set from the dashboard UI) via `make_fragment_scan_exp()`, and
- manually launched fragments from vanilla ARTIQ `EnvExperiment`s using `run_fragment_once()` or `create_and_run_fragment_once()`.

ndscan.experiment.entry_point.make_fragment_scan_exp(*fragment_class: type[ExpFragment]*, **args*, *max_rtio_underflow_retries: int = 3*, *max_transitory_error_retries: int = 10*)→ type[FragmentScanExperiment] [source]

Create a `FragmentScanExperiment` subclass that scans the given `ExpFragment`, ready to be picked up by the ARTIQ explorer/...

This is the default way of creating scan experiments:

```
class MyExpFragment(ExpFragment):
    def build_fragment(self):
        # ...

    def run_once(self):
        # ...
```

```
MyExpFragmentScan = make_fragment_scan_exp(MyExpFragment)
```

class **ndscan.experiment.entry_point.FragmentScanExperiment**(*managers_or_parent*, **args*, ***kwargs*) [source]

Implements possibly (trivial) scans of an `ExpFragment`, with overrides and scan axes as specified by the `PARAMS_ARG_KEY` dataset, and result channels being broadcasted to datasets.

See `make_fragment_scan_exp()` for a convenience method to create subclasses for a specific `ExpFragment`.

build(*fragment_init: Callable[[], ExpFragment]*, *max_rtio_underflow_retries: int = 3*, *max_transitory_error_retries: int = 10*) [source]

Parameters:
- **fragment_init** – Callable to create the top-level `ExpFragment()` instance.
- **max_rtio_underflow_retries** – Number of RTIOUnderflows to tolerate per scan point (by simply trying again) before giving up.
- **max_transitory_error_retries** – Number of transitory errors to tolerate per scan point (by simply trying again) before giving up.

prepare() [source]

Collect parameters to set from both scan axes and simple overrides, and initialise result channels.

run() [source]

The main entry point of the experiment.

This method must be overloaded by the user to implement the main control flow of the experiment.

This method may interact with the hardware.

The experiment may call the scheduler's `pause()` method while in `run()`.

analyze() [source]

Entry point for analyzing the results of the experiment.

This method may be overloaded by the user to implement the analysis phase of the experiment, for example fitting curves.

Splitting this phase from `run()` enables tweaking the analysis algorithm on pre-existing data, and CPU-bound analyses to be run overlapped with the next experiment in a pipelined manner.

This method must not interact with the hardware.

class **ndscan.experiment.entry_point.ScanSpecError** [source]

Raised when the scan specification passed in `PARAMS_ARG_KEY` is not valid for the given fragment.

ndscan.experiment.entry_point.run_fragment_once(*fragment: ExpFragment*, *max_rtio_underflow_retries: int = 3*, *max_transitory_error_retries: int = 10*, *overrides: dict[str, list[tuple[str, ParamStore]]] = {}*) → dict[ResultChannel, Any] [source]

Initialise the passed fragment and run it once, capturing and returning the values from any result channels.

Parameters:
- **max_rtio_error_retries** – Number of times to catch RTIOUnderflow exceptions and retry execution. If exceeded, the exception is re-raised for the caller to handle. If `0`, retrying is disabled entirely.

- **max_transitory_error_retries** – Number of times to catch transitory error exceptions and retry execution. If exceeded, the exception is re-raised for the caller to handle. If `0`, retrying is disabled entirely.
- **overrides** – Any parameter overrides to apply when initialising the fragment; see `Fragment.init_params()`.

Returns: A dictionary mapping `ResultChannel` instances to their values (or `None` if not pushed to).

---

ndscan.experiment.entry_point.create_and_run_fragment_once(*env: HasEnvironment, fragment_class: type[ExpFragment], *args, max_rtio_underflow_retries: int = 3, max_transitory_error_retries: int = 10, **kwargs*) → dict[str, Any] [source]

Create an instance of the passed `ExpFragment` type and runs it once, returning the values pushed to any result channels.

Example:

```
class MyExpFragment(ExpFragment):
    def build_fragment(self):
        # ...
        self.setattr_result("foo")

    def run_once(self):
        # ...

class MyEnvExperiment(EnvExperiment):
    def run(self):
        results = create_and_run_once(self, MyExpFragment)
        print(results["foo"])
```

Parameters:
- **env** – The `HasEnvironment` to use.
- **fragment_class** – The `ExpFragment` class to instantiate.
- **args** – Any arguments to forward to `build_fragment()`.
- **kwargs** – Any keyword arguments to forward to `build_fragment()`.

Returns: A dictionary mapping result channel names to their values (or `None` if not pushed to).

ndscan.results API

General result handling tools

`ndscan.results.tools` module

---

ndscan.results.tools.find_ndscan_roots(*datasets: dict[str, Any]*) → list[str] [source]

Detect ndscan roots among the passed datasets, and returns a list of the name prefixes (e.g. `ndscan.`).

Query and extract parameters used

`ndscan.results.arguments` module

Functions for pretty-printing user argument data (scan parameters, overrides, …) for FragmentScanExperiments from ARTIQ results.

---

ndscan.results.arguments.extract_param_schema(*arguments: dict[str, Any]*) → dict[str, Any] [source]

Extract ndscan parameter data from the given ARTIQ arguments directory.

Parameters: **arguments** – The arguments for an ARTIQ experiment, as e.g. obtained using `oitg.results.load_hdf5_file(…)["expid"]["arguments"]`.

---

ndscan.results.arguments.dump_overrides(*schema: dict[str, Any]*) → Iterable[str] [source]

Format information about overrides as a human-readable string.

Returns: Generator yielding the output line-by-line.

`ndscan.results.arguments.dump_scan`(*schema: dict[str, Any]*) → Iterable[str] [source]

Format information about the configured scan (if any) as a human-readable string.

Returns: Generator yielding the output line-by-line.

`ndscan.results.arguments.summarise`(*schema: dict[str, Any]*) → str [source]

Convenience method returning a combination of `dump_overrides()` and `dump_scan()` ready to be printed.

Quick plotting for notebooks

`ndscan.results.pyplot` module

Previous Next