ETA Documentation

ETA Team

Dec 21, 2021

Contents

Extensible Timetag Analyzer

CHAPTER 1

Introduction

Time resolved measurements are widely used for many research field, including

- Time correlated photon counting
- Fluorescence lifetime imaging
- Anti-bunching
- Linear optics quantum computing
- Quantum process tomography
- Scanning fluorescence microscopy
- Photo-activated localization microscopy (PALM)
- Stochastic optical reconstruction microscopy (STORM)
- Stimulated emission depletion (STED) microscopy

Depending on the duration of the experiments and frequency of events, time resolved measurements can easily generate huge amounts of data, which is overwhelming for common data analysis software.

We attempt to extract the useful information form data generated from time-resolved measurements by introducing a new kind of time-tag analysis software.

ETA, the extensible time tag analyzer, is an event driven programming language with graphical user interface for analyzing, plotting, and fitting of time tagged data.

1.1 Why use ETA?

Fast In order to make ETA fast, our approach is twofold. First, we try to find the fastest algorithm for time tag processing. On the other hand, ETA utilizes LLVM, a state-of-the- art low-level code optimizer, to perform assembly code transformation. This way we generate fast target-specific code for Intel x64 and other processors. ETA allows fast processing of large amounts of time-tagged data, with a scalability from personal computers to super computers.

Flexible ETA allows user defined analysis which is suitable for most of the existing analysis methods, which previously required using different software. ETA is also prepared for upcoming tasks.

User-Friendly ETA provides an easy-to-understand graphical user interface (GUI), with a novel programming paradigm dedicated to simplifying time tag processing.

1.2 Citing ETA

ETA is a result of our scientific research. A Zenodo entry () tracks every released version and can be cited in the Methods section to help readers to identify the specific software version. We discuss the concepts behind ETA in our recent article (Z. Lin et al 2021 JINST 16 T08016) and would appreciate it if you cited it in your work.

CHAPTER 2

Installation

2.1 Installing ETA

ETA is comprised of two parts communicating via websocket: the GUI and the backend. We chose to separate the program in this way to allow remote analysis of time tags. Since correlators are typically close to the setup and require a fast interface for data transfer, it is advisable to run the backend on the computer controlling the correlator. The GUI, however, can be used from anywhere with a morden browser, and network access to the data aquisition computer. If you transfered the time tag files to your own computer for evaluation, make sure you are running both the GUI and the backend locally.

Currently, ETA (>=0.7.0) has been tested on 64-bit versions of Microsoft Windows 7/10, Ubuntu 20.04 and Mac OS 10.15 with Python 3.8/3.9, but it may also work nicely on other platforms. We recommend users to install ETA as a standalone program on Windows, and as a Python package on other platforms.

2.1.1 Install as a standalone program (Windows-only)

You can install ETA GUI and ETA Backend as a standalone program. Currently, Windows x68_64 binary builds are provided on Github.

New installation

- You can download ETA from Github Releases (https://github.com/timetag/ETA/releases). You will need only the `ETA_Install-win64.zip` file for the installer of ETA GUI and ETA Backend.
- Run the extracted `ETA-Setup-x.x.x.exe` to install ETA GUI and ETA Backend. (It is recommended to temporarlily disable realtime thread scanning on your anti-virus software to accelerate the file unzipping.)
- After installation, two icons will be created on the desktop. Click the *ETA Backend* (icon with black background) to start the backend first, and then click *ETA* (icon with white background) to launch the GUI.

Updating the existing installation:

- ETA will attempt to download a new release if one exists at program start. It will then be automatically installed when the program is closed. If you prefrer to do it manually you can run the extracted `ETA-Setup-x.x.x.exe` like in a fresh installation.
- Check the Github Releases (https://github.com/timetag/ETA/releases) for further information about whether the recipes should be updated.

2.1.2 Install as a Python package

For offical Python distribution:

- If you use offical Python installation, type `python3 -m pip install etabackend --upgrade` to install ETA. Please note that on certian platforms you should put `python` instead of `python3`.
- After installation, type `python3 -m etabackend` to start ETA Backend.

For Anaconda/Miniconda in separate environment:

- Open the Anaconda promt and type the following lines. This will create a conda environment for ETA with python newer than 3.8 but not 3.9, yet (incompatible with llvmlite).

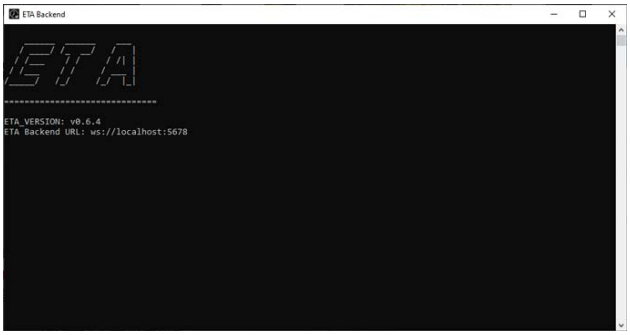
```
conda create -n ETA python=3.8
conda install -n ETA pip
activate ETA
pip install etabackend
```

- Activate the ETA conda envronment, and type `python -m etabackend` to start the ETA Backend.
- Optionally, you can create a short cut for faster launching. Create a text file and add the following, adjusting the path to your Anaconda installation, then save save as a .bat file.

```
@echo off
set PATH=%PATH%;C:\Users\<username>\Anaconda3\Scripts
@call C:\Users\<username>\Anaconda3\Scripts\activate.bat ETA
@call python -m etabackend
```

Launch the GUI

- When the Backend is started, it should look lile



- Open a Web browser window and type the ETA URL on the address bar to launch the GUI.

2.1.3 Verify the Installation

- After launching the ETA GUI, click New and then click Simulation, it will load a simulation recipe which can run without any actual timetag. Click the "Run" besides the Display Panel you want to execute to see the results.
- In order to analyze your own time tag file, you can drag a recipe (`recipe_name.eta`) onto the main screen to load it, specify filename in the variables, and then click "Run" button besides the Display Panel you want to execute.

If problem occurs, try `pip install --upgrade` again to upgrade each ETA dependencies .

2.2 Configurating ETA for remote access

The computer which runs the ETA Backend will do the number crunching and it might be advantageous to run it on the computer which has the timetag files so the (potentially large) files do not have to be copied around. This is just a recommendation, though.

- In the ETA GUI you can then specify the IP address and port number of the computer running the backend, which can just be `localhost:5678` if you run backend and frontend on the same PC with the default settings.
- Remote access is disabled by default to keep your computer safe. You can enable it by setting the enviroment variable `ETA_LISTEN` to `0.0.0.0`, and set `ETA_IP` and `ETA_PORT` to the IP address and the port that you want to use for remote connection to this computer.

Note: Remote access might be dangerous if your IP and port are directly accessible via Internet. Considering setting up a firewall.

2.3 Try ETA without installation

You can access the ETA Web GUI on any platform with a Web browser.

Note: Please note that the Web GUI will always follow the current master branch, and it may not match your ETA Backend version.

CHAPTER 3

Using the pre-made recipes

Here we want to provide some information about the pre-made recipes.

3.1 Simple Lifetime

This recipe is determining time differences between events on two different channels. This means starting the stopwatch on an event on one channel and stopping on an event on another channel. These time differences are then logged into a histogram. Depending on the `ch_sel` (read: channel select) variable it can record differences between the `SYNC` and the delayed channel 0 (`t1`), between the `SYNC` the delayed channel 1 (`t2`) or between the delayed channels 0 and 1 (`t3`).

3.1.1 Input

You can browse your file system to select a timetagged file using the folder icon on the right of the `file` variable. If the filename is removed afterwards, leaving only a folder as `file` input, the code will run over all the files in the given folder.

3.1.2 Delay

The channels are duplicated (0->2 and 1->3) with a configurable delay by the dealy line instruments. Afterwards the actual time differences are taken from the delayed copies. To adjust the delay, change the second argument of the `emit(channel_number, delay_in_ps)` function. This feature lets you compensate for physical differences in fiber patchcord and coaxial cable length for the two optical and electrical signal paths in an HBT setup.

3.1.3 With or without reset

A Start-Stop measurement can be performed with or without resetting the start time. To understand the difference, imagine the starting of the clock being triggered by an event on channel 0. If the next event happens to occur on channel 1 everything is clear: the clock is stopped and the time difference is recorded to the histogram. But if the second event

occurs on channel 0 we have to decide how to handle this case. We can either ignore this event and all other events on channel 0 until an event shows up on channel 1 (start-stop without reset) or we can reset the clock on every event on channel 0, only measuring the shortest time differences (start-stop with reset). The latter case approximates a proper correlation at time delays close to 0 and is typically used as an analysis if full correlation is not available. Note, that for long time scales it is absolutely neccesarry to do a full correlation to get an accurate result, especially if interresting features are not at 0 time delay (e.g. if the antibunching dip in an autocorrelation of single photons is shifted away from 0 time delay). You can switch between reset modes by adding/removing a transition from the `running` state to itself with the channel number of the start channel.

3.1.4 Output

You can choose to display an interactive histogram plot with plotly or pyplot or you can save a `*.txt`, `*.png` and `*.eps` file by running the corresponding code panel.

3.2 Correlation

This recipe is determining the time differences between all events on the zeroeth and first physical channel, starting the time on each event on channel 0 and stopping on each event on channel 1 for each start (full correlation). These time differences are then logged into a histogram.

3.2.1 Input

You can browse your file system to select a timetagged file using the folder icon on the right of the `file` variable. If the filename is removed afterwards, leaving only a folder as `file` input, the code will run over all the files in the given folder.

3.2.2 Delay

The channels are duplicated (0->2 and 1->3) with a configurable delay by the dealy line instruments. Afterwards the actual time differences are taken from the delayed copies. To adjust the delay, change the second argument of the `emit(channel_number, delay_in_ps)` function. This feature lets you compensate for physical differences in fiber patchcord and coaxial cable length for the two optical and electrical signal paths in an HBT setup.

3.2.3 When to use

It should be the first recipe you think about for auto- and cross correlation. It is also the right choice for investigating features far away from 0 time delay.

3.2.4 Output

You can choose to display an interactive histogram plot with plotly or you can save a `*.txt` file by running the corresponding code panel. You can also have a look at the examples for fitting in the interactive plot and the examples for saving figures of a zoom-in and a full auto-correlation including analysis.

3.3 Two-time correlation

This recipe records the time differences between a sync channel and two other channels into a 2D histogram. A `COINCIDENCE(name, slots, emit_on_ch_x_when_filled)` tool is used to record stops for events on the two channels. If ch 0 has two or more events before ch 1 has an event, the most recent timestamp is used, like in the case of start-stop with reset. Once both slots of the `COINCIDENCE` tool are filled, it emits a signal on a virtual channel to let the program know that it is time to record the time differences to the last sync before the earlier event into a 2D histogram.

3.3.1 Usage

Since the plot is 2D the amount of data ponts can increase dramatically with small changes in the bin-size and bin number. The current plotting libraries used cannot handle these amounts of data easily and it is therefore advised to keep the bin number < 500 and increase the histogram range by binning with the binsize. The data can be moved around in the 2D histogram by adjusting the third element in each of the two dimensions of the `HISTOGRAM(name,[(number_of_bins,bin_size,"time-your_delay"),(number_of_bins,bin_size,"time-your_delay")])`.

3.3.2 Output

You can choose to display an interactive histogram plot with plotly or bokeh or you can save a `*.eps` and a `*.png` file by running the corresponding code panel.

3.4 Realtime

This recipe allows you to view an on-the-fly analysis of your data while the correlator is still recording it. This has been tested with HydraHarp and quTAG correlators. You can either accumulate a histogram or only show the latest update e.g. for alignment. The analysis performed in this example is a full correlation.

3.4.1 Usage

As explained in the start-stop and correlation recipes, you can adjust a delay if the feature you are interested in is at the edge or outside the histogram area. You can do this by opening the "Instrument Designer" for the delay lines `DL0-2` and `DL1-3`. The delay lines copy the events on channel 0 (or 1 in case of `DL1-3`) to a new channel (first argument of the emit function) with a delay specified in the second argument of the `emit` function: `emit(new channel number, delay in picoseconds)`. You can adjust the width of the histogram. This is done with a combination of the `bins` and `binsize` variables in the start screen. The y-axis will automatically rescale to accomodate the growing histogram. An important adjustment is the speed at which the file is processed. This is done by selecting how many events one chunk should have before the program bothers to analyse this chunk. Use the variable `records_per_cut` on the home screen to adjust this. (note: this will happen automatically soon)

You might want to switch between accumulating the histogram to showing only the most recent chunk. We call these modes accumulation and alignment mode, respectively. By default the graph will start accumulating the histogram but a button can switch to alignment mode if desired. There is also a button to switch between logarithmic and linear plotting of the y-axis.

3.5 Frequently asked questions

3.5.1 How are the channel number assigned in ETA?

In order to easily migrate recipes between different time tagger hardware, ETA unifies the physical channels and virtual channels in the same "address space". See RFILE in Tools.

3.5.2 How is HHT3 mode different from other modes?

Assuming the sync channel (CHN0) is always connected to the laser and CHN1 and CHN2 are connected to SPDs, there should be no difference on the recipe side. There will be some measurement error only if the sync is not perfectly stable.

However, if you plug SPDs signals on CHN0 and CHN1, then you will need different recipes for T2 and T3, since CHN0 is not recorded in T3 mode, and we need special actions for guessing the last sync signal. Transitions on ETA graph cannot be triggered on the non-existing CHN0 signals in the T3 mode.

3.5.3 Graph looks strange sometimes

See https://github.com/timetag/ETA/issues/59

CHAPTER 4

Making recipes

Each ETA recipes consitis of `Virtual Instruments` , `Script Panel` and optionally `Parameter`.

Here is the workflow of building an ETA recipe, if you are not using ready-made ones.

4.1 Add Virtual Instruments to perform analysis

Click the `Virtual Instruments` button on the main screen to create a new analysis routine and open it in the Instrument Designer.

This is the place where you define how exactly the ETA backend analyses your data.

You create an analysis instrument in two steps:

- Create a state diagram (https://en.wikipedia.org/wiki/State_diagram) through which the program transitions that covers all relevant states that your analysis undergoes (left-hand side of the instrument)
- Create some (simple) code to define what actions should be performed upon one (or several) specific transitions (right-hand side of the Instrument Designer)
- Add your time-tagger in Virtual Instruments

Make an RFILE in the action panel.

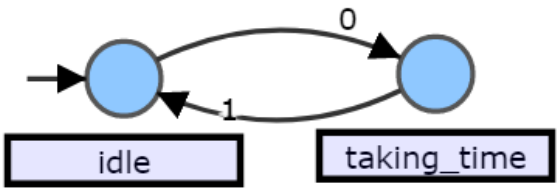
Specify the name for RFILE, assign the physically available channels and the number of marker channels.

TODO: explain how to do it with PicoQuant HydraHarp400

- We can use the state diagram described above to analyze a time tag file with two channels in a start-stop manner. For this we need to add a histogram into which we save the time differences between events. We also need a clock to record these time differences. Both these entities can be created with the help of the "Create" menu in the top bar of the Instrument Designer. You can also directly type into the code panel:

  ```
  HISTOGRAM(name, (number_of_bins, bin_size))
  ```

  ```
  CLOCK(name)
  ```



From this point on I will assume that the state diagram is labelled as follows:

I will also assume the histogram is named h1 and the clock is named c1. We will define actions so that we use channel 0 as the start channel and channel 1 as the stop channel. (Note, that this analysis will not record time differences between closest events, since the start is not reset if a second event occurs on channel 0 before an event occurs on channel 1. See Section "Coincidence Measurements")

To define an action you select a transition after which you would like the action to happen.

With this transition selected press SHIFT + T (think: Trigger). You will see state_at_arrow_tail–list_of_channel_numbers–>state_at_arrow_head followed by a colon (:) appear in the code on the right-hand side. By using indentations you can now specify actions that should be performed upon completion of the transition. In case of a start-stop measurement we want to start the clock when there is an event on channel 0. We therefore write:

```
idle--0-->taking_time:
  c1.start()
```

To stop the clock and record the time difference in our histogram we write:

```
taking_time--1-->idle:
  c1.stop()
  h1.record(c1)
```

Additional Info:

- States can loop to themselves.
- Labels can be written underneath the state (e.g. when they become too long to fit) with SHIFT + M (think: Mark)

TODO:explain the following and add more functions

```
COINCIDENCE()
TABLE()
```

Allowed action definitions

TODO: Insert graph

```
a--1-->b:
  action1
a--2,4-->b:
  action2
b: #involves all transitions arriving to b
  action3
```

TODO: explain all the analysis actions

```
start(clock)
start(c1)
Start a clock labeled c1.

stop(clock)
stop(c1)
Stop a clock labeled c1.

emit(channel_number,waittime_ps,period_ps,repeat)
emit(2,10)
Emit signal to a channel 2 after 10 picoseconds.

record(histogram,clock)
record(h1,c1)
Record the time interval on clock c1 to histogram h1.

fill(coincidence,slot_number)
fill(coinci1,1)
Record coincidence event on slot 1 of coincidence tool coinci1.

clear(coincidence,slot_number)
clear(coinci1,1)
Clear coincidence event of coincidence tool coinci1.
```

4.2 Add Script Panel

In the `Script Panel` you tell ETA to run your analysis and define what happens with the result.

A minimum example that saves the data as an Origin-compatitable `*.txt` file looks as follows:

```
import numpy as np
result =eta.run(eta.clip(filename)) #tell ETA to run the analysis on "filename"
histogram = result["h1"] #get the table from result
np.savetxt("h1.txt",histogram) #save the txt file for the histogram
eta.send("processing succeed.") #display message on GUI popup
```

Instead or in addition to saving a file, the data can be displayed/treated in various ways. In the following example dash from plotly is used to create an interactive graph from a histogram. `app` is a Dash object which gets modified with the style configurations. `eta.display(app)` is used for displaying the Dash on the GUI side.

```
import numpy as np
import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly.graph_objs as go
result =eta.run(eta.clip(filename))
histogram = result["h1"] #get the table from result

app = dash.Dash()
app.layout = html.Div(children=[
    html.H1(children='Result from ETA'),
    html.P(children='+inf_bin={}'.format(inf)),
    dcc.Graph(
        id='example-graph',
        figure={
```

(continues on next page)

(continued from previous page)

```
            'data': [
                {'x': np.arange(histogram.size), 'y': histogram, 'type': 'bar', 'name
↪': 'SF'},
            ],
            'layout': {
                'title': expname
            }
        }
    )
])

eta.display(app)
```

Please refer to our pre-made recipes for inspiration.

4.3 Run your analysis

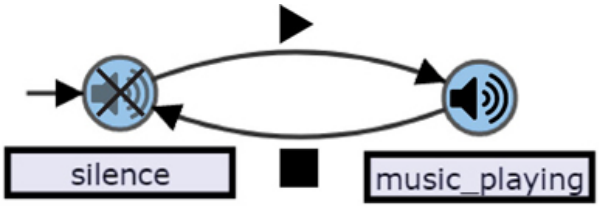
Once you have added `Virtual Instruments` and `Script Panel`, return to the home screen and press `Run` on the `Script Panel` of your choice.

CHAPTER 5

State Diagram

To get a better understanding, let's first consider a simple example of a state diagram with two state before we move on to a working ETA example.

The states we want to consider for our abstract example are `music_playing` and `silence`. A transition between the states is triggered by either `play` or `stop`, depending on the current state.



We start in the `silence` state, indicated by the arrow with it's tail attached nowhere.

If we want this to become musical chairs we need to define some actions. We would like participants to start running around the chairs when entering the `music_playing` state and sit down on the chairs when entering the `silence` state, then remove the participant who did not find a chair and remove one of the chairs.

- We can create a similar diagram to the example above by left-clicking to create a state, left-clicking again to create a second state and left-click-dragging between the states to create the transitions. Or you can left-click-drag on the first state to directly create the second state and a connecting transition and then create the remaining transition by left-click-dragging back from the second state to the first one. It is important for the system to have a defined starting point. We can define the initial state by selecting a state (left-click) and then pressing SHIFT + I (think: Initial). To label a transition with its condition, select a transition and double click it. Transition labels must be channel numbers separated by commas (0,1,2) with channel numbering starting at zero (0).

  The labelling mode for states can be entered the same way but names can be any string of allowed characters (alphanumeric and most special characters, but not spaces and commas).

  All states and transitions must be labelled.

CHAPTER 6

Tools and Actions

As mentioned before, Virtual Instruments are used to perform timetag analysis in an ETA recipe, and each Virtual Instrument consists of a graph on the left-hand side and a code panel on the right-hand side. Tools and Actions can be put in the code panel to specify what should be done when there is a state change on the graph.

Each Action belongs to a certain Tool. Tools can be created with a user-specified name and some other parameters. The name is used to refer to the Tool later when performing Actions. The parameters that have default values can be omitted.

For example, if you want to record a time interval of two events, you can create a CLOCK Tool called `clock` first, and then do Action clock.start() to start this clock.

In the following documentation, we list the built-in Tools and their Actions in the current version of ETA.

Note: Please note that ETA is still under development. Tools and their Actions might be changed in the future.

6.1 RFILE

```
RFILE(name, [signal channels], [marker channels])
```

RFILE is a read-only source of timetags. It works like a placeholder, for the physical timetagger device in the real-world, in the Virtual Instruments. The availble channels in the physical device will be mapped to the assigned channel number, which can be used by all the Virtual Instruments.

Each RFILE should be later connected with a Clip generator in `eta.run`, see *Customizing Script Panel* for more details.

6.1.1 Parameters

- **`signal channels` (default:[0,1,2,3])** This parameter specifies the signal channels mapped from the timetag source. `[0,1,...]` reads as `[first real channel, second real channel, .`

`..]`. You should always put *ALL* availble channels on the timetagger device in the list, even if some of the channels wasn't plugged-in.

- **`marker channels` (default:[])** Simialr to `signal channels`, but this parameter specifies the marker channels. This parameter is only for HydraHarp devices.

Note: The RFILE Tool can be defined on any Virtual Instrument graph. You just need to define it once, and it works as if the the signals are emitted the signal from that Virtual Instrument.

Channel numbers in RFILE should be continously ascending, like `[1,2,3]` or `[2,3,4]`, and any of them should be smaller than any virtual channel number. There should be a clear boundary between virtual channel numbers and real channel (signal and markers channel) numbers.

6.2 CLOCK

`CLOCK(name, max start times, max stop times)`

CLOCK is a time interval recorder with a start button and a stop button. The CLOCK remembers the time when it is started or stopped, and it calculates the time interval as the output.

There is a special case of CLOCK called "multi-CLOCK", where the buttons can be pressed for multiple times. The "multi-CLOCK" has a maximum limit of recorded events, and it will drop the oldest event when it reaches this limit.

Note: Please also note that ETA will not automatically clear the recordings in the CLOCK, they can only be overwritten.

6.2.1 Parameters

- **`Max Start Times` (default:1)** This parameter specifies the maximum limit of start events that can be stored in the recorder.
- **`Max Stop Times` (default:1)** This parameter specifies the maximum limit of stop events that can be stored in the recorder.

Note: Max Stop Times should be set to 1, and Max Start Times should be set to a very large number when doing correlational measurements, so that you correlate each signal on one channel to every signal on the other channel.

The performance will not be influenced significantly if this value is very large, since only part of the recording of the clock, say `c1`, will actually read during `h1.record(c1)`, which is decided by the histogram size at compiling time.

Auto-expanding of this valued **was removed** for performance consideration. Because expanding will make the analysis stuck completely everytime it moves old recording to a new continuous memeory, and a linked-list based solution is not suitable here as it breaks the nice caching for analysis actions that favors continuous memeory, like `h1.record(c1)`.

6.2.2 Actions

- **`clock.start()`** Start the clock at the current time.
- **`clock.stop()`** Stop the clock at the current time.

- **`clock.start(LAST_SYNC)`** Start the clock at the last sync. The time for the last sync is calculated from SYNCRate.
- **`clock.infer_start_from_stop(SYNC)`** Using the stopping time to find the last specified type of signal before it, and then overwrite the starting time to the time of this signal. If the clock is a single-start-multi-stop clock, then the earliest stopping time value it stores will be used for inferring the start.

  Note: `clock.infer_start_from_stop(SYNC)` and `clock.start(LAST_SYNC)` use the SYNC period to reconstruct the SYNC signal, which is not recorded in HydraHarp T3 mode files.

  If multiple T3 mode file are used, they will both starts at time 0. If the SYNC rates are the same, they are automatically synced without extra efforts. If the SYNC are different, the SYNC will be taken from the first defined `RFILE`, as the master `RFILE`. You can also mannualy set SYNC rate to a clip using `your_t3clip.SYNCRate_pspr` in the Script Panel, stretching all files' internal time to match the master `RFILE`.

  If T3 and T2 are mixed in sources, it should work similarly. Things get a little bit complicated if T2 doesn't start at 0 while T3 does. In order to mix them properly, you will need to manually modify the T2 Clip object with a negative value added to `your_t2clip.GlobalTimeShift`, which will applied to all channels within the that Clip.

- **`[clock1,clock2,...].sort(start)`**

  Sort the starting time of a group of clocks, preserving their stopping time. This is useful if you want to record multi-dimensional histograms with the axis indicating the arrival order (first photon, second photon) instead of channels (detector1, detector2).

  Note: Please note that multi-CLOCK is not yet supported.

  The first parameters can be also changed to `stop`, to sort the stopping time of a group of clocks, preserving their starting time.

6.2.3 Examples

Performing a start-stop measurement:

```
CLOCK(c1,1,1)
state2:
    c1.start()
state1:
    c1.stop()
    t1.record(c1)
```

6.3 HISTOGRAM

```
HISTOGRAM(`name`,[(`number_of_bins`, `width_of_bins_in_picoseconds`,
`time_interval_modifier`),...], [`image_pixels_x`, `image_pixels_y`])
```

Histogram generates statistics of time intervals. The time intervals that fall out of the histogram will be ignored. Histograms can be 1-dimensional or multi-dimensional, and it can be put into an array that is 1-d or multi-d.

The histogram can be retrieved using its name from the returned dictionary from eta.run() in the Script Panel for further processing and plotting.

6.3.1 Parameters

- **[(`number_of_bins`, `width_of_bins_in_picoseconds`, `time_interval_modifier`),...] (requ**
  A list of dimension specification. One tuple is used per dimension. The first value of the tuple indicates the number of bins in the histogram. The second value of the tuple indicates the size of each bin in the histogram.

  The third value of the tuple is optional, it is a string of numba expression wrapped by `""` that modifies the time intervala. It should take a variable of the old time interval named `time` as the input, and returns the new time interval, which will be used later for locating the index of bins in this dimension.

Note: The product of the histogram parameters bin size and bin number gives you the maximum correlation length, if you are performing a correlational analysis.

If the histogram is multi-dimensional, specify one tuple for each dimension, like `[(100,16),(200,16)]`.

Note: By default, the Nth bin in the histogram with bin width `binsize` includes the `N*binsize` and excludes the `(N+1)*binsize`. Time interval modifier would be handy if you want want to flip this default policy. For example, if you have 16ps as binsize, and you want to exclude `0ps, 16ps, ...` and include `16ps, 32ps,...` form the `fisrt, second, ...` bins, simply put `HISTOGRAM(h1, num_of_bins,16,"time-1")`.

If you need logarithmic binning, use `HISTOGRAM(h1, [(100,24,"round(math.log(time))")])`. The code actually works as if the time interval modifier is injected to every `h1.record()` throughout the recipe.

If you need a super long linear histogram that exceeds the memory, try making a histogram with the time interval modifier `"time-`histogramoffset`"`. This would move the position of "time zero", thus truncate the histogram to a given position from left. Then you can set different the histogramoffet with Parameter on the main GUI, or from the Script Panel. You may run the same analysis many times with the same timetag file source but different `histogramoffset`, and glue the histogram results together on a disk.

- **Extra Dimensions** The 1-d or multi-d histogram can be put into an array which is 1-d or multi-d, forming "an array of (consistent) histograms" The extra dimension adds before the histogram dimensions, usually used for making images.

6.3.2 Actions

- **histogram.record(clock)** Record the time interval of `clock` into a 1-dimensional histogram.
- **histogram.record(clock1,clock2,...)** Record the time interval of `clock1` and `clock2` into a multi-dimensional histogram. The order of clock should be the same as the order of dimension.

  This is usually used to explore the joint probability distribution of two types of events.
- **histogram.record_all(clock)** `histogram.record_all` is the Cartesian product version of `histogram.record`. It records all the time intervals of the multi-CLOCK `clock` into the histogram.

  This is usually used together with a multi-start-single-stop CLOCK to correlate signals from one channel to the another channel.

  Using record_all with a multi-dimensional histogram is not yet supported.

- **`histogram[x][y][...].record(clock,...)`** Record the time interval of `clock` into an image of 1-dimensional or multi-dimensional histograms. This is usually used in biology imaging, where x and y can be obtained from state transitions of markers, which indicates the position of the scanning instrument.
- **`histogram[x][y][...].record_all(clock,...)`** Combination of `histogram[x][y].record(clock,...)` and `histogram.record_all(clock)`
- **`histogram.clear()`** Zero-out the histogram. Useful for making a histogram of a given period of time.

6.3.3 Examples

Performing a correlation:

```
HISTOGRAM(t1,(100,16))
CLOCK(c1,100,1)
started:
    c1.start()
stopped:
    c1.stop()
    t1.record_all(c1)
```

6.4 COINCIDENCE

`COINCIDENCE(name, num_of_slots, coincidence_flag, time_interval_threshold)`

Coincidence is a Tool that keeps track of a coincidence. It has multiple Coincidence Slots which can be filled with timetags individually. The coincidence condition is fulfilled, when all of the slots are filled and the time interval between the current time and each of slots is less or equal to `time_interval_threshold`. This tool will create an INTEGER with `coincidence_flag` to indicate if the coincidence condition is fulfilled.

6.4.1 Parameters

- **`num_of_slots` (required)** The number of coincidence slots in this coincidence tool.
- **`coincidence_flag` (required)** An INTEGER to indicate if the coincidence condition is fulfilled.
- **`time_interval_threshold` (default: INF)** Time interval in picoseconds between the current time and each of slots should be less or equal to `time_interval_threshold` to fulfill a coincidence.

6.4.2 Actions

- **`coincidence.fill(slotid)`** Fill (or overwrite) the coincidence slot *slotid* with the current time. Then it checks immediately if the coincidence condition is fulfilled, and changes the INETGER `coincidence_flag` to either 0 or 1 as an indication.
- **`coincidence.clear(slotid)`** Clear the coincidence slots *slotid*.
- **`coincidence.reset()`** Clear all coincidence slots.

6.4.3 Examples

Usually, you would like to fill different slots at the events from different input channels.

You may also want to do a conditional emittion when the coincidence condition is fulfilled, to generate singals for further analysis like counting the number of coincidences in other VIs.

```
COINCIDENCE(co1, 2, co1_flag)
a--6-->a: #trigger on signal form chn6
    co1.fill(0) # fill the slot 0 of the coincidence tool, co1_flag may flip␣
↪automatically according to the coincidence condition
    emit(12,0,0, co1_flag) # emit on chn12 only when co1_flag==1
    co1.reset() # clear all the coincidence slots
a--7-->a: #trigger on signal form chn7
    co1.fill(1) # fill the slot 1 of the coincidence tool, co1_flag may flip␣
↪automatically
    emit(12,0,0, co1_flag) # the same conditional emission.
    co1.reset()
```

6.5 INTEGER

`INTEGER(int_name, init_value)`

Integer is a Tool that records a integer value. It will be shared across actions, embedded code, and it will be returned in the results.

6.5.1 Parameters

- **`init_value` (default: 0)** The initial value for the integer that will be assigend right after starting a new analysis, before feeding in the fisrt Clip.

6.5.2 Actions

- **`int_name=<liter_value>`** Assign value to an INTEGER tool. You can also use it in the embedded code.

6.6 SELF

The instrument itself is also a Tool. When using its actions, the instrument doesn't need to be referred by its name.

6.6.1 Actions

- **`emit(chn, waittime=0, period=0, repeat=1)`** Emit a signal to `chn` after `waittime`, both are either integer values or the name of an INTEGER Tool. It can also emit some repeated signals with a *period* in picoseconds if `repeat` is set to larger than one.

  The maximum limit of channel number `chn` is 255, and the minimum limit of `chn` is larger than the largest channel number assigned for the `RFILE`.

Note: It is not allowed to emit to any channel in a `RFILE`, since it is read from a timetag file (timetagger channels or markers). The emitted signal will never be written to `RFILE` to prevent corrputing the read-only timetag data.

You can use a INTEGER as a flag to do a conditional emittion. `emit(8, 0, 0, co1_flag)` will only emit on channel 8 when `co1_flag == 1`. When `co1_flag==0` no event will be emitted.

If you need to merge signals from two channels into one channel, simply emit them into a new unused virtual channel. Channels can also be used as routers. For examples, you can route events to different Virtual Instruments based on some status that is controlled by the markers.

- **`cancel_emit(chn)`** Flush all the previously emitted events in the channel `chn`.

  Cancelling emitting a real channel from a timetag file will terminate the analysis before the ETA reaches the ending of the current section of the file.

- **`interrupt()`** Pause the current analysis and return to Python code in the Script Panel, if auto-feed is disabled.

  This is useful for implementing time-based clipping or ROI (region-of-interest) clipping.

  You can use `interrupt()` to pause the analysis, when a certian event happens or a certian state is reached. Then, from the Python side you can get the current positions for every timetag Clips, which was from multiple time taggers and provided in `eta.run()` as sources. With those positions, you can later run analysis using some new clips constructed form the same file.

  Note: Please note that `interrupt()` will **NOT** do anything if auto-feed is enabled. Set `max_autofeed=1` to disable it.

  For time-based clipping or ROI (region-of-interest) clipping, when `interrupt()` happens, `result,task=eta.run({'timetagger1':generator1,...},return_task=True, max_autofeed=1,...)` will return, so that you can get absoulte positions of the original timetag file using `your_pos1 = result['timetagger1'].get_pos()`.

  Discard `result` if you are performing time-based cutting, and use the histograms in the `result` to decide if you would need to perform `clip.get_pos()` if you are doing ROI cutting. You may also need to keep the `task` discriptor, if you want to resume this paused analysis to find the second cutting point. Then you can either truncate the original timetag files into many small ones, or save this absolute position list for later use with `eta.clip_file(...,read_records=your_pos2-your_pos1, seek_record=your_pos1)`

  Please note that ROI cutting and time-based cutting should be viewed as advanced alternative to `eta.split_file()` and `eta.clips`. In most of the cases, you can built a event router with conditional `emit()`, which can be easily integerated into existing analysis and run in realtime.

- **`abort()`** Abort the analysis and return to Python code in the Script Panel, leaving the results at their current states. Unlike `interrupt()`, the analysis task can neither be resumed by auto-feed nor by manual resumption.

6.6.2 Examples

Making a delay line:

```
ch0_event:
    emit(2, 130)# here, ch0 is duplicated to ch2 with a delay of 130ps, making a␣
↪delay line
```

(continues on next page)

(continued from previous page)

6.7 Extending Actions using Embedded Code

Apart from the built-in actions, you can also use a embedded code to extend the functionality of ETA.

Embedded code can be wrapped in a `{` and `}` . If the code contains curly brackets, a pair of `{{{` and `}}}` can be used.

The embedded code uses a restricted sub-set of Python language, and a limited subset of Numpy function is imported with *np*. Internally, ETA uses Numba to compile the Python code into LLVM and link it with the built-in actions and other parts of the program.

Note: Please note that features that requires `import`, `exec` or file I/O are not available. Calling built-in actions in embedded code is not currently supported.

We noticed that the built-in tools and actions already serve as a good basis for many different experiments. But we still want to add more actions for different analysis purposes. If you created some custom Action for extending the functionality of ETA, please share it :)

6.7.1 Examples

Here is an example for sampling randomly delays from a exponential decay and then emitting a signal with this delay whenever a transition from state a to state b happens via channel 1. This type of embedded code might, for example, be useful for a Monte-Carlo simulation.

```
INTEGER(random_delay) # define an INTEGER for use by both actions and embedded Python
a--1-->b:
    #execute the embedded Python code
    {
        delay_from_sync = 200
        binsize = 16
        random_delay_arr = ((np.random.exponential(125, 1)+delay_from_sync)/binsize)
        random_delay = round(random_delay_arr[0])*binsize
    }
    emit(3,random_delay) # emit on the channel 3 with a dealy of random_delay
```

Here is an advanced example for simulating a 50%-50% beam splitter for randomly redistributing a singal on channel 3 to channels 4 and 5.

```
VFILE(4)
VFILE(5)
INTEGER(retchn)
a--3-->b:
    {
        options = np.asarray([4,5])
        retchn = np.random.choice(options)
    }
    emit(retchn,0)
```

CHAPTER 7

Script Panel

In the ETA recipe, Script Panel provides the user interface for each experiment.

The common usage of Script Panel is to start an analysis on an existing time-tag file or a realtime time-tag sources, display results, and provide interactive controls of the acquisition device (time-tagger) or the analysis.

ETA provides Python API to allow customization of the Script Panel. Here we list all the API in the latest version of ETA. Please refer to the pre-made recipes for examples.

Note: When running the Script Panel, the entire recipe will be sent to ETA Backend. ETA will first check the Virtual Instruments and then start executing the code for the current Script Panel.

The code is executed in an isolated environment with an `eta` object and other user-defined parameters (only those in the same group of Script Panel will be visible) as global variables.

You can also import third-party Python libraries like `numpy` or `matplotlib`, etc.

Please note that ETA Backend can only run one Script Panel at the same time. If you have multiple ETA GUI connected to ETA Backend, the running requests will be put in a queue.

7.1 Prepare Timetag Clips

You need to prepare Clips of time-tags before running the analysis. Clips contian the actual timetag data, together with the metadata of the timetag data, like measurement resolutions, which are crucial to the correctness of an analysis. Clips can be loaded from a section of timetag file, or constructed from an exsiting buffer given by the timetagger library.

Due to the nature of the timetag data, we recommend using the Python generator function that yields Clips each time when it is called. ETA will automatically feed the new Clips from the generator when a current Clip reaches its end. ETA provides a set of APIs to help you build generators that automatically cut the file into a series of Clips. These APIs will read the header of the time-tag file and then set the Clip objects up, which can be later used in the analysis.

For users who are not familiar with generators, we use Python generator to implement the `eta.clips()` generator function, as a warpper of `eta.clip_file()`. And the `eta.split_file()` generator function will further warp the `eta.clips`.

You can also implement your own generator using Clip class or a lower-level API *eta.clip_file*. One example is that you can poll a timetagger library to see if there is new records in memory ready for analysis, when performing ETA streaming analysis.

7.1.1 eta.clip_file(filename, modify_clip=None, read_events=0, seek_event=-1, format=-1, wait_timeout=0)

`eta.clip_file` takes the previous Clip and number of events to be read as input, and generates a new Clip by sliding the cut window in the time tag file.

Note: The low-level API, *eta.clip_file()*, is the only way to actually load the timetag from file into memory and return only one Clip object for later use. You can think of *eta.clip_file()* as the timetag-specific way of doing `read()` in Python.

However, it is not recommended to use this low-level API directly, as it is complecated to manage multiple Clips during analysis. In order to keep the analysis running on a series of Clips, you will also need to put *eta.clip_file()* and *eta.run* together inside a *for* loop, with a task variable managed by yourself.

- **`filename`** The path to the time tag file. Please note that if you run ETA Backend on a remote computer, you need to specify the path to file on that computer.
- **`modify_clip`** If this parameter is None, ETA will read the file header and try to construct a new Clip object. If provided, ETA will modifed this previous Clip instead of creating a new Clip object from scratch by reading the header again, for better performance.

  Note: When used together with `seek_event=-1`, ETA will slide the a window with length `read_events` in the time-tag file, starting from the ending position of the prevoius Clip. By iteratively feeding the returned Clip as `modify_clip`, one can fetch newly generated events from the file. It is useful for implementing real-time analysis, and there is a higher-level API, `eta.clips`, which does this automatically for you.

  After calling this function, the `modify_clip` will be updated with timetag events from a new window in the timetag file. If you would like to keep the old Clip, please make a deep copy of the Clip object before calling this function.

  If you would like to read a header-less file of a supported format, you could mannualy construct an empy Clip object with required format information, and feed that as modify_clip. Refer to Advanced Usage for more ideas.

- **`read_events`** The number of desired events to be loaded into the returned Clip. Setting it to 0 will make ETA read the entire file.

  If the number exceeds the size of the time-tag file, and `wait_timeout` is configured, ETA will wait until the file grows to that size. This is particularly useful in a real-time analysis where the time-tag file is continouslys growing.

  If file failed to grow to the desired size, a shortened Clip to the current ending of the file will be loaded. If no records can be loaded at all, a False will be returned indicating a failure, and the `modify_clip` will not be modified.

Note: You can also set a negative value, then the number of records in this Clip will be calculated as the number of records between the ending of the last Clip to the current end of file minus the absolute value of this negative number.

The time tag file that serves as the FIFO when you perform a real-time analysis might have pre-allocated-but-not-yet-written areas, and the negative value here can help you get rid of that.

- **`seek_event`** Setting the starting event number for reading. Setting to it `0` will force ETA to read from the first event after the file header.
- **`wait_timeout`** Value in seconds specifies the maximum waiting time. ETA will wait until the file grows to desired size. If file failed to grow to the desired size, a shortened Clip to the current ending of the file will be loaded.
- **`format`** Format specifies the time-tag file format that you want to use in the analysis. The default is set to the auto detection of PicoQuant devices. You can also use the constant `eta.FORMAT_SI_16bytes` for Swabian Instrument binary format, `eta.FORMAT_QT_BINARY` for qutools quTAG 10-byte Binary format, `eta.FORMAT_QT_COMPRESSED` for compressed qutools quTAG binary format, or `eta.FORMAT_BH_spc_4bytes` for Becker & Hickl SPC-134/144/154/830 format, or `eta.FORMAT_ET_A033` for Eventech ET A033 format.

Note: The format of time-tag you use might influence the analysis results in unexpected ways, through the nature of timing system it uses.

If the timetag file is recorded with absolute timing (default for most of the time taggers), then every cut should keep the same absolute timing.

If the timetag file is recorded with relative timing (like in HHT3 mode), then the absolute timing for each cut will take the first event in this cut as the reference of zero. You should be extremely careful when using `seek_event` to seek to arbitrary position, as the file format supports only continuous sequential read.

7.1.2 eta.clips(filename, modify_clip=None, read_events=1024*1024*10, seek_event=-1, format=-1, wait_timeout=0, reuse_clips=True, keep_indexes=None)

`eta.clips` makes a generator that yields Clips with a specified amount of new record read from the file. It is wrapper on top of *eta.clip_file()*. Instead of returning only one Clip object, it will return a generator that yields a Clip every time it called. It inherts most of the parameters from *eta.clip_file()*, and also adds some new parameters.

- **`read_events`** This amount of events will be read for each Clip that this generator yields.
- **`seek_event`** Setting the starting event number for reading the first clip. This parameter will be ignored starting from the second clip, as the second second clip would slide a window of `read_events`. If you want to skip some windows, use `keep_indexes` instead.
- **`reuse_clips`** If set to False, the previous Clip will not be modifed, and a new Clip will be created everytime it is called.

Note: This is useful when you want to load all the Clips at once. For example, in a correlational analysis, we can set this parameter to False, and then use `list(ret)` to load the file into some equal-size Clips in a list, with which you could run parallel analysis to get speed boosts.

Please be careful when setting this to False, as it may cause memory leaking if the references are not handeled properly.

- **keep_indexes** A list of indexes of the sliding windows for Clips that will be actually yielded. Other Clips will be skipped. Indexes start from 0 for the first window of `[0,read_events]`, and index 1 means [read_events,read_events*2].

7.1.3 eta.clips_list(filename, read_events=1024*1024, format=-1, threads=os.cpu_count()*2)

`eta.clips_list` makes a list of generators using `eta.clips` for parallel analysis. It sets the `keep_indexes` or `seek_event` automatically for each of the generators, so that the file is splited into roughly equal sizes for each generators.

Unlike `eta.split_file`, which makes one single generator that splits the file into a certain number of equal size Clips and yield them one by one, `eta.clips_list` would return a list of generators, and each of them can `eta.clips` the file into a configurable `read_events` size.

- **threads** How many threads you want to use. It will decide how many sections you want to split the file into, and how many clips generators are returned.

7.1.4 eta.split_file(filename, modify_clip=None, cuts=1, format=-1, wait_timeout=0, reuse_clips=True, keep_indexes=None)

DEPRECATED `eta.split_file` is simple wrapper on top of *eta.clips()*, that makes a generator that splits the file into a desired amount of equal size Clips. It inherts most of the parameters from *eta.clips()*.

- **cuts** The number of Clips that you want to generate. Default value is set to 1, thus the full time-tag will be returned in one cut descriptor.
- **keep_indexes** A list of indexes of the sliding windows for Clips that will be actually yielded. Other Clips will be skipped. Indexes start from 0 for the first window of `[0,read_events]`, and index 1 means [read_events,read_events*2].

  Examples:

```
#stop evaluation of timetag stream after 2%
cutfile = eta.split_file(file,100,keep_indexes=[1,2])
result = eta.run(cutfile)
```

7.2 Executing Analysis

7.2.1 eta.run(sources, resume_task=None, group="main", return_task=False, return_results=True, background=None, max_autofeed=0, stop_with_source=True)

`eta.run()` starts an analysis, where you actually feed all sources into RFILES in Virtual Instruments and obtain results.

You can use Python generators functions, that yields Clip objects, as a source. ETA will do auto-feeding , fetching one new Clip from the generator each time, so the generator functions will be called many times.

In a single invoke of `eta.run()`, only a single task will be used for all Clips generated by the generator, until the generator reaches its end or `max_autofeed` is reached. By default `eta.run` will use a new task for the analysis, unless `resume_task` is specified.

The analysis will block the execution of Python script until the **results are returned in a Python dictionary, with the ''HISTOGRAM'' names as the keys**. If you want to schedule many analysis and run them in parallel, you can set `return_results=Fasle`. Clip objects indicating the current reading positions for all generators in the sources, will also be returned in the results dictionary with the `RFILE` names as keys.

- **`sources`** A dict of Python generators functions that yields Clips. The keys should match with the name of coresponding RFILEs in the virtual instrument.

  If only one generator is provided instead of a dict, it will be distributed to all RFILEs, which might cause unexpected behaviors.

- **`max_autofeed`** It limits the number of Clips that `eta.run` would fetch from the generator.

  Set this value to 1 if you want to get each result for every single Clip from the generator, rather than get final result after the full generator is consumed.

- **`background`** Run the analysis in the background. Set it to True will start a new therad in the thread pool. By default `background=not return_results`

  In this case, you must turn on `return_task` so that the task descriptor will be returned immediately, and the analysis will continue running in the background. You can start many threads in the background and gather a list of task descriptors, with which you can aggregate the results from these threads later.

- **`return_results`** Specifies if a dictionary of results should be returned.

  This is the switch for multi-threading analysis. No new thread will be created and the analysis will be performed in MainThread if this option is set to True.

  Note: The parameter for enabling multi-thread mode is removed since version 0.6.6, when we switch to the Map-Reduce style of multi-threading. The new way of doing multi-threading is easier and more flexibile. `eta.run` works like Map, and `eta.aggregrate` works like Reduce.

  You can schedule your analysis from Script Panel in any way you want. As long as you keep the task descriptor, you will be able to retrieve the result in the end.

- **`return_task`** Specifies if the modified task descriptor should returned.

  You must set it to True if `return_results` is set to False. If both of them are set to Ture, you can get both of them with `result, task = eta.run(..., return_task=True, return_results=True)`, and later you can resume an analysis with the task descriptor using `resume_task`.

  Note: The context parameter is renamed to task descriptor to reduce confusion since version 0.6.6.

  Task descriptor works like a complete memory snapshot of a current running or finished analysis. Everything except for the sources (Clips) is preserved. If you want to reprod can resume the analysis later without worrying about underlying details of the anlaysis.

- **`group`** The group name of instruments that you want to run analysis on. This parameter is provided so that `eta.run` can automatically call `eta.create_task` using the provided group name when `resume_task` is not provided.

- **`resume_task`** Specifies an existing task descriptor to resume the analysis.

A new task descriptor can be created with `eta.create_task`. You can also iteratively call `eta.run` using the returned task from a previous `eta.run` call. In second case, the analysis will be resumed from the point where it ended, with all contexts set correctly, and then feeded with the new Clip. This is particularly useful when you want to perform real-time or streamming analysis.

Note: After the analysis is resumed, the old task descriptor becomes invalid, however, a new task descriptor can be returned by setting `return_task=True`.

The way how the files is cut into Clips, or the order in which `eta.run` is invoked, will never affect analysis result, as long as you always resume with the last task descriptor (or `None` for the first iteration) during the entire analysis.

In multi-threading analysis, however, there will usually be the same amount of "last" task descriptors missing during the fisrt iteration, as the number of threads you use. You will also end up with that amount of task descriptor in the end. For some analysis, like correlation which yields histograms, you can use `eta.aggregrate` later to merge the analysis results from those tasks into one. But it won't change the fact that they are essentialy many different independent analysis.

- **`stop_with_source`** Stop the analysis when any of the sources reaches its end. Set it to False if you want to run simulation without any source.

7.2.2 eta.create_task(group, resume_task=None):

`eta.create_task` will create a new analysis task using the group of instruments. The returned task can be used in `eta.run(resume_task=task)` The instruments within the same group are visible to each other in this task.

- **`group`** The group name of instruments that you want to run analysis on.
- **`resume_task`** Same as in `eta.run()`.

7.2.3 eta.aggregrate(list_of_tasks, sum_results=True, include_timing=False):

`eta.aggregrate` will gather data form previous multi-threading anlaysis tasks started with `return_results=False` and put them together as the final results. If all previously anlaysis tasks haven't finished, ETA will block until all of them are finished.

- **`list_of_tasks`** A list of previously created task descriptors, from which you want to retrieve results.

Note: You can run multi-threading analysis on different groups for completely different analysis at the same time. However, you can only aggregrate the results using form task descriptors created by `eta.run` on the same group.

- **`sum_results`** Specifies if the results will be summed up.

Note: This is useful for correlational analysis if you want to merges histograms from many individual analysis tasks. Keep in mind that you will need to make sure that it is physically meaningful to perform adding. (Is the histogram in the same base unit? Can you add histograms from experiments done today and yesterday? Will the result be different from running with with only one task, but many Clips instead.)

Users can also set this value to False and get a list of dict returned instead. Then they can use their own data aggregation methods, like concatenating to generate large images.

- **include_timing** Specifies if the timing information `eta_total_time`, `eta_compute_time`, `max_eta_total_time`, `max_eta_compute_time` should be appended into the results.

  Examples:

```
clipgens=eta.clips_list(file)
# assign different index_range to different clip generators, so that␣
↪they read different parts of the original file
tasks = []
for cutfile in clipgens:
    tasks.append( eta.run({'UniBuf1':cutfile}, group='compile',
↪return_task=True,return_results=False))
    # start a thread in the background for each clip generator
    # and keep the reference to the task descriptor
results = eta.aggregrate(tasks,include_timing=True)
```

7.3 Interacting with ETA GUI

7.3.1 eta.display(app)

You can send results to ETA GUI using this function. The value of app can be either a Dash or Bokeh graph currently.

Note: Use `app = dash.Dash()` to create a Dash graph.

7.3.2 logging.getLogger('etabackend.frontend')

Returns the `logger`, with which you can display inforamtion on the GUI.

7.3.3 logger.info(msgstring)

This is the ETA alternative for `print()` in Python. This is useful when you use want to display some message on the ETA GUI.

- **msgstring** Message string to be shown on the ETA GUI.

7.3.4 logger.setLevel(loglevel)

This modifies the logging level of a specific logger.

- **loglevel** A loglevel from logging. Can be `logging.WARNING`

  Examples:

```
logger = logging.getLogger('etabackend.frontend')
logger.info('No further logoutput for the realtime recipe.')
logger.setLevel(logging.WARNING)
plt.show()
logger.setLevel(logging.INFO)
```

7.3.5 eta.send(text,endpoint="log"):

This is useful when you want to talk to another program other than ETA GUI via WebSocket (see Advanced Usages). You can stream the results back using this function.

- **`text`** String of information to be sent to the client.
- **`endpoint`** Can either be `log` or `err`, for indicating the type of message.

7.4 Modify recipies programatically

You can also modify recipes programmatically. The recipe uploaded will be availble under `eta.recipe`.

As an example, you can upload the template recipe from your LabVIEW program to ETA Backend via WebSocket (see Advanced Usages), and then change the parameters (like bin size for histograms) to get different results.

7.4.1 eta.load_recipe(jsonobj=None)

Converting the `jsonobj` to a Recipe object and save it into eta.recipe. Then refresh compling cache if eta.recipe is modified.

- **`jsonobj`** A JSON object parsed from `.eta` file. If not provided, the current `eta.recipe` will not be modifed.

7.4.2 eta.recipe.get_parameter(name)

Get the value of a parameter in the recipe, given the name of the parameter. If there are multiple parameters with the same name, only the first one will be returned.

- **`name`** Name of the parameter, as shown in the ETA GUI.

7.4.3 eta.recipe.set_parameter(name, value=None, group=None)

Set the value (and group) of parameters in the recipe, given the name of the parameters.

- **`name`** Name of the parameter, as shown in the ETA GUI.
- **`value`** The new value of this parameter. If None is given, the value will not be modified.
- **`group`** The new group of this parameter. If None is given, the group will not be modified.

Note: The updated parameters will be applied to the next Run. Call `eta.load_recipe()` after finishing updating parameters and before `eta.run` if you want to apply it immediately.

7.4.4 eta.recipe.del_parameter(name)

Delete a parameter in the recipe, given the name of the parameter. If there are multiple parameters with the same name, only the first one will be deleted.

- **`name`** Name of the parameter, as shown in the ETA GUI.

7.5 Using Third-party Libraries

The following libraries are required to be installed with ETA. Feel free to use them in your recipes.

- numpy
- scipy
- lmfit
- matplotlib
- dash
- dash-renderer
- dash-html-components
- dash-core-components
- plotly
- bokeh

Using other third-party libraries (including Python libraries or dynamic linked libraries) might lead to not fully portable recipes. Please distribute the libraries with the recipe, so that the users can download and install them. ETA also recommends distributing the libraries on ETA-DLC (ETA downloadable contents).

CHAPTER 8

Advanced Usage

8.1 Construct or modify the Clip object mannualy

You may want to make a Clip object without using `eta.clip_file` or other higher-level APIs built upon this. For example, you may want to build a Clip from a memory buffer recived form a remote computer over the Internet. You may also want to read a header-less file of any supported format, by skipping the default header-reading behaviors.

8.1.1 Read a header-less file

Here is a sample to create a empty Clip object in the script panel, which could be fed into `eta.clips` as `modify_clip` and so that `eta.clips` will not read the file header at all.

```
import etabackend.clip as clip
first_clip = clip.Clip()
first_clip.TTRes_pspr = int(TTRes) # you will need to mannualy configure TTRes␣
↪(integer in picoseconds). It was usually read from the header
first_clip.DTRes_pspr = int(DTRes) # same for DTRes
first_clip.SYNCRate_pspr = int(SYNCRate) # same for SYNCRate
first_clip.BytesofRecords = 4 # for PicoHarp, HydraHarp, and TimeHarp
first_clip.RecordType = 0x00010203 # 0x00010203 for PicoHarp T2 mode,  0x00010303 for␣
↪T3 mode.
# By using ret_clip = eta.clip_file(filename) to read a complete file (with header)␣
↪generated by the program from the instrument vendor, you can find the correct value␣
↪of ret_clip.RecordType and ret_clip.BytesofRecords.

my_clip = eta.clips(filename,... , modify_clip=first_clip)
eta.run(... my_clip ...)
```

8.1.2 Build a Clip without any file

Here is an example to avoid `eta.clip_file` or other higher-level APIs that reads a file on disk, by creating the Clip directly from the memory buffer.

```
import etabackend.clip as clip
clip_from_remote_pc = clip.Clip()
clip_from_remote_pc.TTRes_pspr = ...
clip_from_remote_pc.DTRes_pspr = ...
clip_from_remote_pc.SYNCRate_pspr = ...
clip_from_remote_pc.BytesofRecords = ...
clip_from_remote_pc.RecordType = ...
your_buffer, read_length = your_network_lib.recv(1000)
clip_from_remote_pc.buffer = your_buffer
clip_from_remote_pc.batch_actualread_length = read_length
clip_from_remote_pc.next_RecID_in_batch = 0
```

Internally, ETA performs zero-copy analysis with the given buffer. This example could help if you have throughput issues with the hard-drive and want to direclty perfrom the analysis with the memory buffer obtained by the instrument driver program over USB, PCIE, or the Internet.

Noet that this exmaple is for only one device and it's over-simplified. It's suggested to use the implementation of `eta.clips` as a reference to properly warp your devices as generators, so that ETA could properly handel the synchronization between multiple devices.

8.1.3 Applying global time shift

You may also want to modify the Clip returned by `eta.clip_file` or other higher-level APIs to apply a global time shift to all channels within a certain timetag file. This is handy if you use multiple timetag as sources (RFILES) in one experiment. Refer to `clock.infer_start_from_stop` for more ideas.

```
ret_clip = eta.clip_file(filename)
ret_clip.GlobalTimeShift = -1,000,000 # picoseconds
new_clip = eta.clips(filename, seek_event=0, modify_clip=ret_clip, reuse_clips=True)
# use seek_event=0 to resume to the first event after the header
# make sure reuse_clips=True, so that your modification will be preserved when the␣
↪generator is sliding windows
eta.run(... new_clip ...)
```

8.2 Run ETA as a Python Library

There are two ways to run ETA as a Python Library, one with the `BACKEND` Class and the other with ETA Class.

Use the `BACKEND` Class if you want full ETA Backend features, without Websocket and GUI. This is ideal for using ETA in monitor-less (headless) enviroments like supercomputers, or embedded devices.

Use the `ETA` Class, if you would like to ignore all Script Panels in the exsiting recipe and simply obtian a `eta` object for later use, as described in *Customizing Script Panel*. This is ideal for performing automated testing, using ETA with a notebook environment like Jupyter, or integrating ETA into your own Remote Procedure Calling system.

8.2.1 backend.process_eta(recipe, id, group="main")

Run a Script Panel, as if it is being run from the GUI. You will usually need to hook a `send` function to obtian results, as the Script Panel code might use logger or other methods to stream the results to the caller.

- **`recipe`** The recipe object parsed from the `.eta` JSON file.
- **`id`** The identifier of the Script Panel to be started.

- **group** The gruop name of this Script Panel

```
import json
from etabackend.backend import BACKEND
backend = BACKEND(run_forever=False)
def send(self, text, endpoint="log"):
    print(text)
backend.send = send
with open("./Realtime.eta", 'r') as filehandle:
    backend.process_eta(json.load(filehandle), id="dpp_template_code", group="main")
```

8.2.2 eta.compile_eta(recipe)

Compile the recipe and cache it in the ETA kernel. You can later call `eta.run` as if in the Script Panel.

- **recipe** The recipe object parsed from the `.eta` JSON file.

  Please refer to the tests for examples.

8.3 Talking to ETA backend via WebSocket

ETA backend implements a Remote Procedure Call mechanism using JSON format, with which you can upload an existing recipe, modifying parameters like `filename`, run the analysis, and even get the real-time streaming of the result.

Before invoking a remote procedure, connect your program (client) to ETA backend (server) using the Websocket protocal.

(Examples in LabVIEW and Javascript are provided. [TODO:link to .vi])

Sending a JSON string in a format of `{"method": "<name of method>", "args": [<arg1>, <arg2>,...] }` to the Websocket will invoke the corresponding procedure immediately. When invoked procedure is running, new requests will be queued until the current one finishes.

The procedure might send JSON strings as responses in a format of `["<type>","<content>"]`. Please note that the client might get multiple responses (even in different types) after invoking a single procedure.

8.3.1 Remote procedures provided by ETA Backend

There are three special functions provided for remote controlling ETA Backend.

All these methods bundle a set of internal functions that first update the recipe on ETA Backend to the uploaded one, and then perform the requested actions. Optionaly they will also send the updated table for GUI as responses. There might be some extra response, for errors in the recipe or user-defined frontend logger in the Script Panel code.

It is not recommended to remotely call the undocumented procedures provided by the backend object, because they are not designed for remote calling and the returned value will not be streamed back to caller's side.

1. VI Checking

   JSON: `{ 'method': "compile_eta", 'args': [eta_file_content] }`

   Arg: *eta_file_content* is a string of the content of the *.eta* recipe.

2. Browse file and set it as the parameter.

JSON: `{ 'method': "recipe_set_filename", 'args': [eta_file_content, id, name] }`

Arg: *eta_file_content* is a string of the content of the *.eta* recipe. For specifying the parameter that you want to modify, the *id* and *name* should also be provided.

3. Run a Display Panel

   JSON: `{ 'method': "process_eta", 'args': [eta_file_content, id, group] }`

   Arg: *eta_file_content* is a string of the content of the *.eta* recipe. For specifying the Display Panel that you want to run, the *id* and *group* should also be provided.

   Extra Responses: Other responses are sent in code of Display Panel in the recipe, using *eta.send()*.

8.3.2 Type of responses from ETA Backend

In order to interact with the Backend properly, your client needs to handel these types of responses, and display them to the user.

1. Errors

   Type: `err`

   JSON: `["err","<text>"]`

   Args: `<text>` is a string of the error message.

2. Generic Information

   Type: `log`

   JSON: `["log","<text>"]`

   Args: `<text>` is a string of the message.

3. Update Main Table

   Type: `table`

   JSON: `["table","<json>"]`

   Args: `<json>` is a JSON string of the main table.

4. Switch state to running

   Type: `running`

   JSON: `["running"]`

5. Switch state to stopped

   Type: `stopped`

   JSON: `["stopped"]`

6. Switch state to discarded

   Type: `discard`

   JSON: `["discard"]`

7. URL of dashboard

Type: `dash`

JSON: `["dash",<url>]`

Args: `<url>` is a string of URL to the dashboard.

8. User-defined message (eg. streamming histogram or GUI updates)

   Type: defined using eta.send(message,type)

   JSON: `["<type>","<message>"]`

   Args: `<message>` is a string of a user-defined message.