

Software Extensions

Extensions are Python modules that enhance the debugging process by providing powerful analysis tools on top of your data. There are currently three types of extensions available:

- High-Level Analyzers
- Analog Measurements
- Digital Measurements

You can browse available extensions and quickly install them from our Extensions Marketplace by clicking the Extensions button located at the right side of the software. You can also publish your own creation! Over time, we'll be able to build up a collection of extensions. We can't wait to see what kinds of extensions will be shared there.

The screenshot displays the 'Extensions Marketplace' interface. On the left, a detailed view of the 'Analog Fundamental Frequency' extension is shown, featuring a Pusheen cat icon, the author 'Mark "Cool Guy" Garrison', version '0.0.3', and an 'Install' button. The description explains that it uses `numpy.correlate` to find the fundamental frequency of the selected analog range. Below the text is a waveform plot. On the right, a sidebar titled 'Extensions' lists several other extensions, each with an 'INSTALL' button:

- Analog Fundamental Frequency** by Mark "Cool Guy" Garrison
- Average Period** by Tim Reyes
- Baud rate estimate** by Jonathan Reichelt Gjensen
- BQ25150 Linear Battery Charger** by Lena Voytek
- Burst Stats** by Peter Jaquiere

Extensions Marketplace

High Level Analyzers

High level analyzers are protocol analyzers that process the output of the existing "low level" analyzers already in the app. You can write your own in Python. This lets you create

powerful new analyzers without needing to reinvent the wheel. So far, our favorite application of high level analyzers is converting existing decoded I2C bytes into clean, decoded messages specific to the I2C device we're working with, to easily read recorded I2C traffic without needing to go back to the datasheet.

PAGE

High-Level Analyzer (HLA) Extensions

>

Digital and Analog Measurements

Digital and analog measurement extensions let you write python code that processes a selected range of analog or digital data, and produces metrics. For example, if you would like to calculate the jitter in a digital clock recording, You can simply write a python script which iterates over the transitions in the selected range, and computes the deviation from nominal. Once written, just shift+click a region of a digital channel, and your measurement result will appear in the list with the other measurements!

PAGE

Measurement Extensions

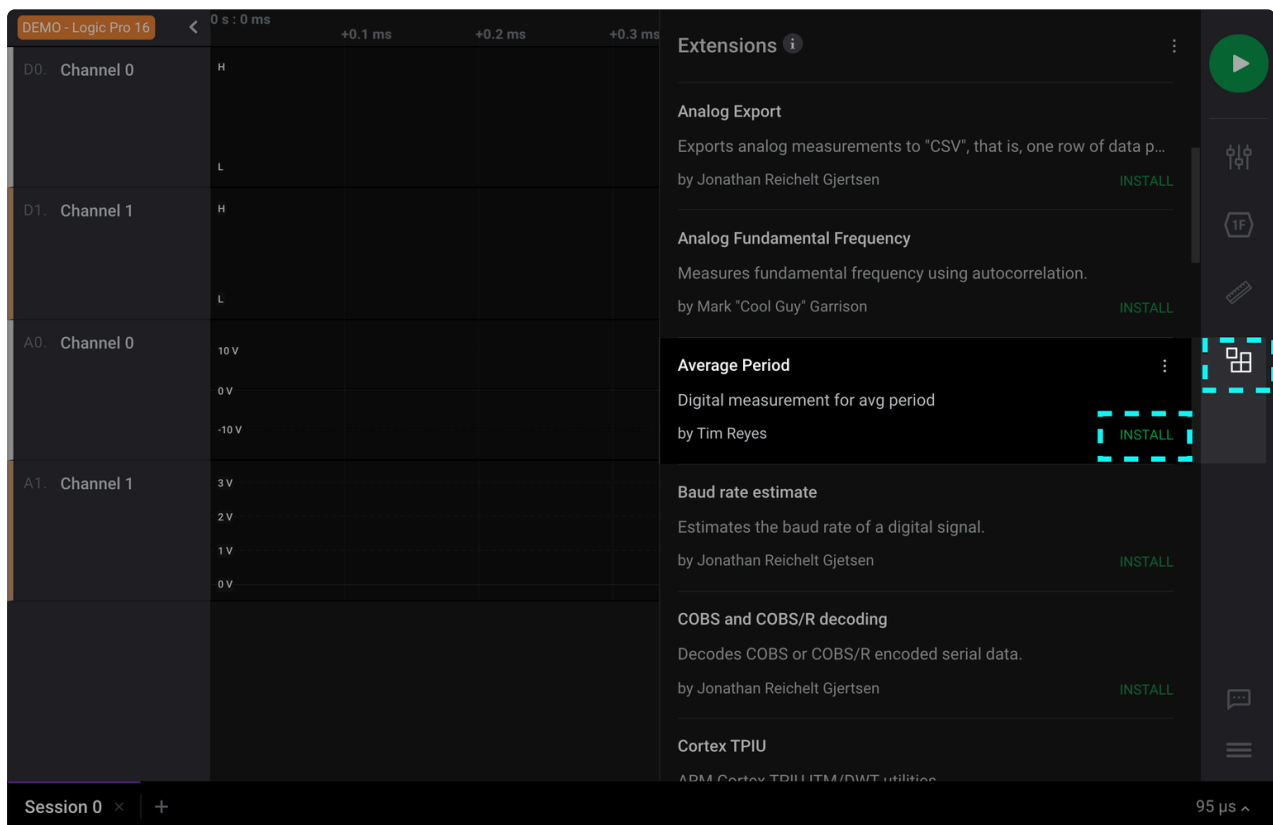
>

Extension Installation

There are 2 methods to installing an extension. You can install an extension from within the software via our in-app Extensions Marketplace, or you can install it via local source files on your PC.

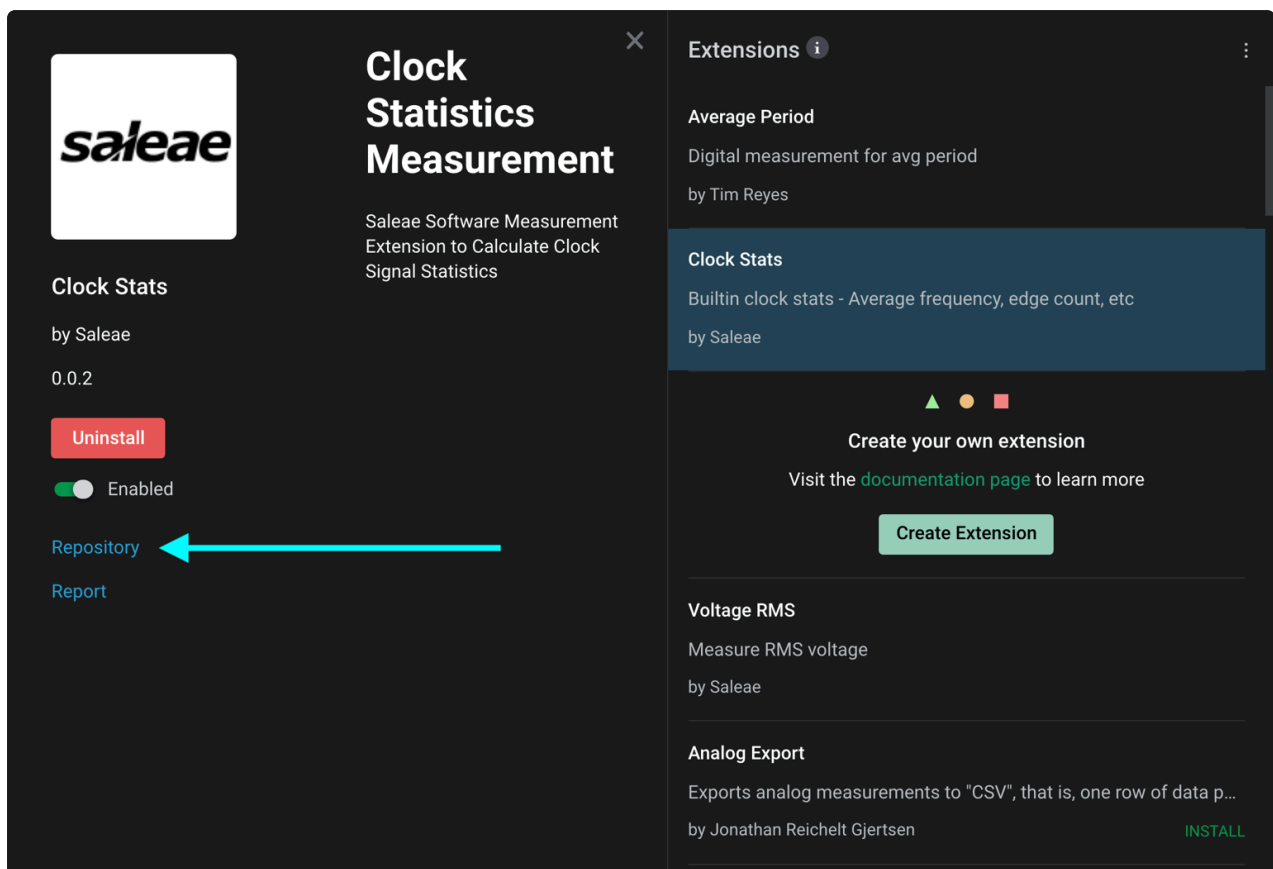
Extension Installation within the Software

This will be the easiest method of installing an extension. Clicking on the Extensions button on the right will open up a list of published extensions from us and from the community. Afterwards, you can click "Install" for the extension of your choice.

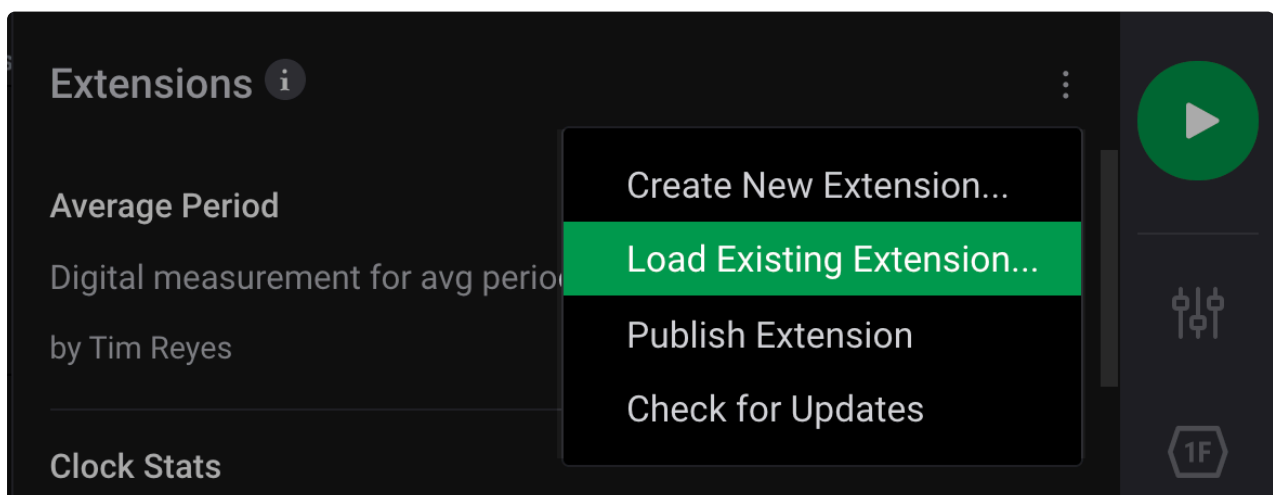


Install Extensions Manually

In some cases, the extensions list may not appear properly within the Extensions panel of the software. This may be due to security settings such as a firewall, or simply having the PC offline. To get around this, extensions can be found on GitHub and can be downloaded manually using the following steps.

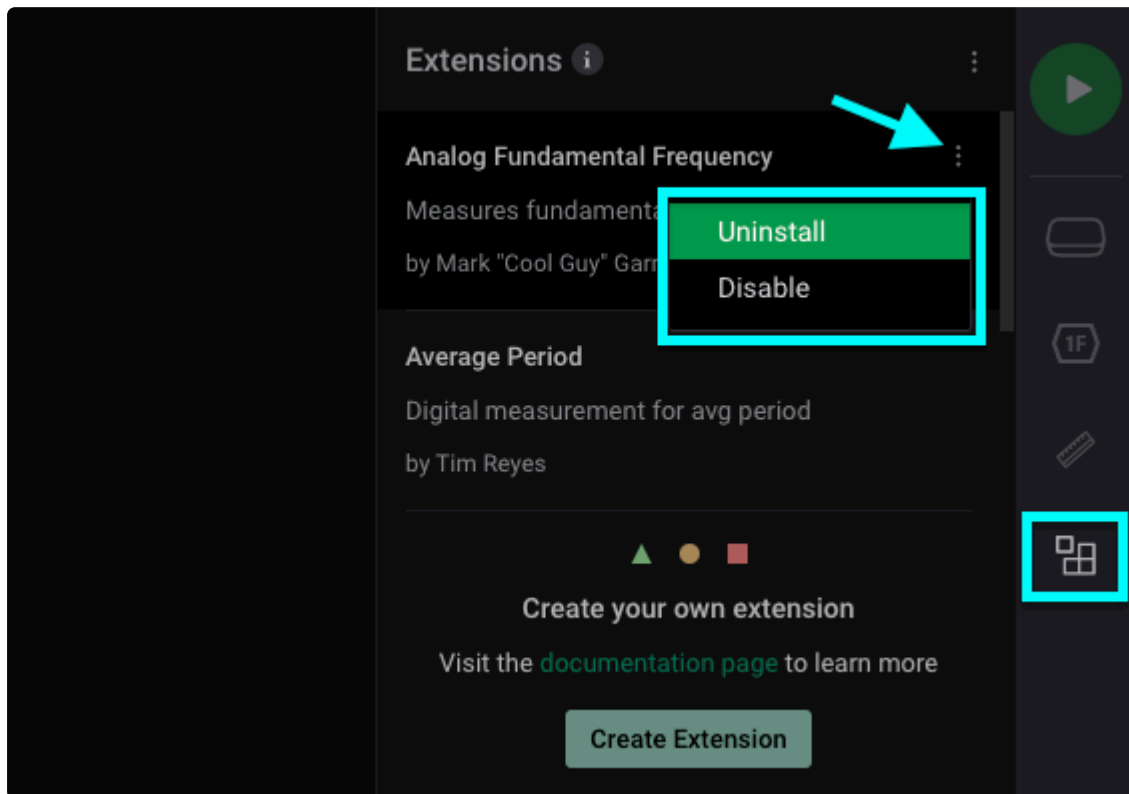


1. On a computer where the marketplace loads, click the extension of interest so that it is highlighted. This will open up the extension information within the software.
2. Click the "Repository" link. This opens the github.com repository for that extension.
3. Click the green "Code" button, and select download as zip.
4. Extract the zip file anywhere.
5. In the Logic 2 software, on the Extensions panel, click the "three-dots" menu icon, select "Load Existing Extension..." and select the downloaded extension.json file.



Uninstalling an Extension

To uninstall an extension, click on the 3 dots next to an extension in the Extensions panel. Then click Uninstall.



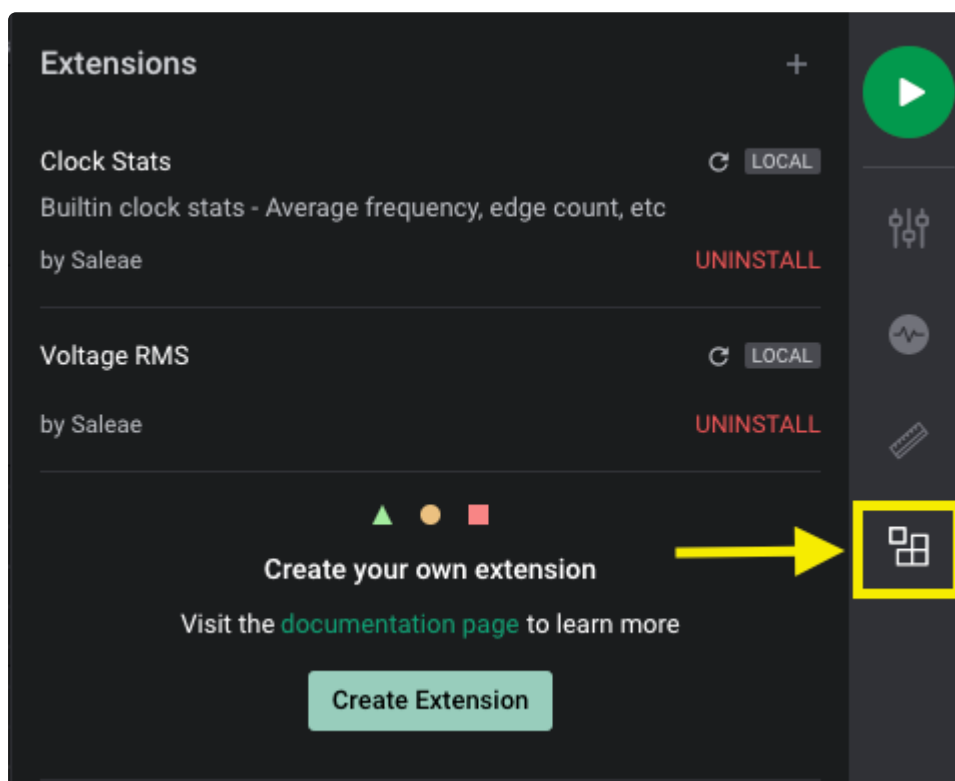
Uninstalling an Extension

Create and Use Extensions

Creating an Extension

In this guide, we will walk your through creating an HLA (High Level Analyzer) extension, however, the process is identical for other types.

1. Click the 'Extensions' panel button on the right of the software



2. Click 'Create Extension'
3. Under 'Generate from template', choose the type of extension you would like to create. For this example, we will create a High Level Analyzer.
4. **(Optional)** Click 'Additional Information' to fill in information about your extension.

Add Extension [X]

Generate from template

High Level Analyzer [v] **Save As...**

▼ Additional Information

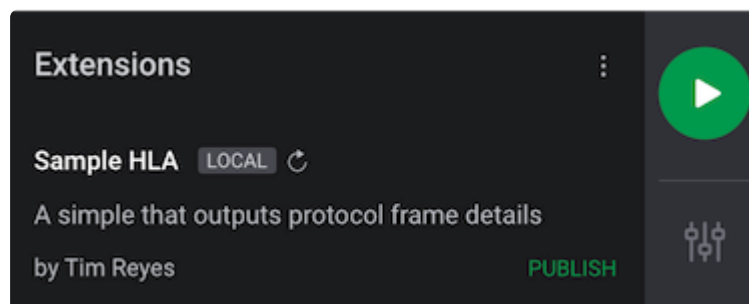
Sample HLA

Tim Reyes

A simple that outputs protocol frame details

[Documentation and examples](#)

5. Click 'Save As...' to save and select your location.
6. You should now see your new extension listed as 'Local' in the software.



Using a High Level Analyzer Extension

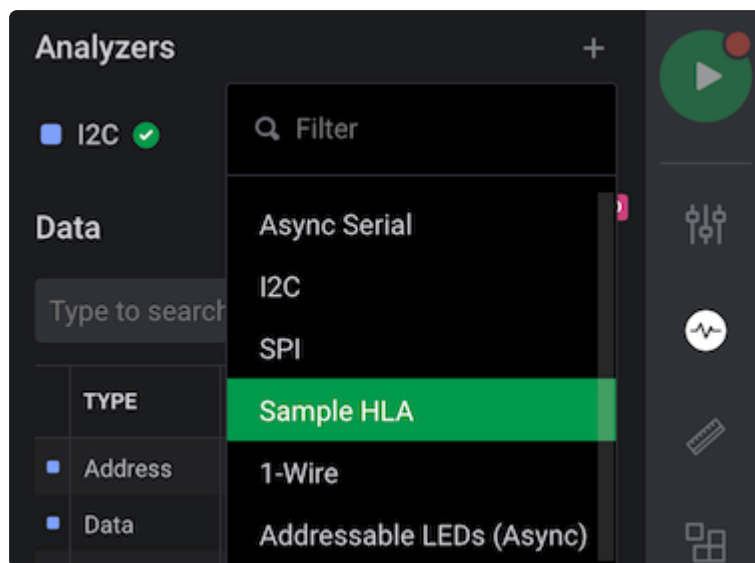
1. To test the new Sample HLA, capture any protocol data for [one of the supported analyzers](#), and add the appropriate protocol analyzer. We've provided an I2C capture below in case you don't have a protocol data recording available.



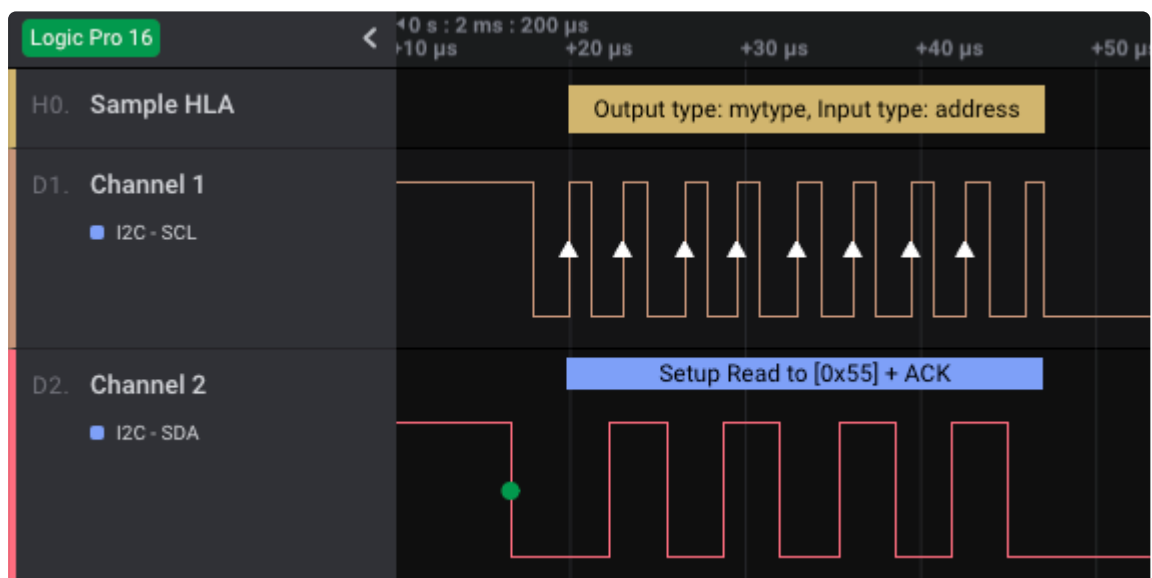
6KB

I2C.sal

2. Click the Analyzers '+' button to add our Sample HLA.



3. In the settings popup, select 'I2C' under Input Analyzer. For the rest of the settings, you can leave them as default and click 'Finish'. Once you add the HLA, you can see it as a virtual channel as shown in the image below.



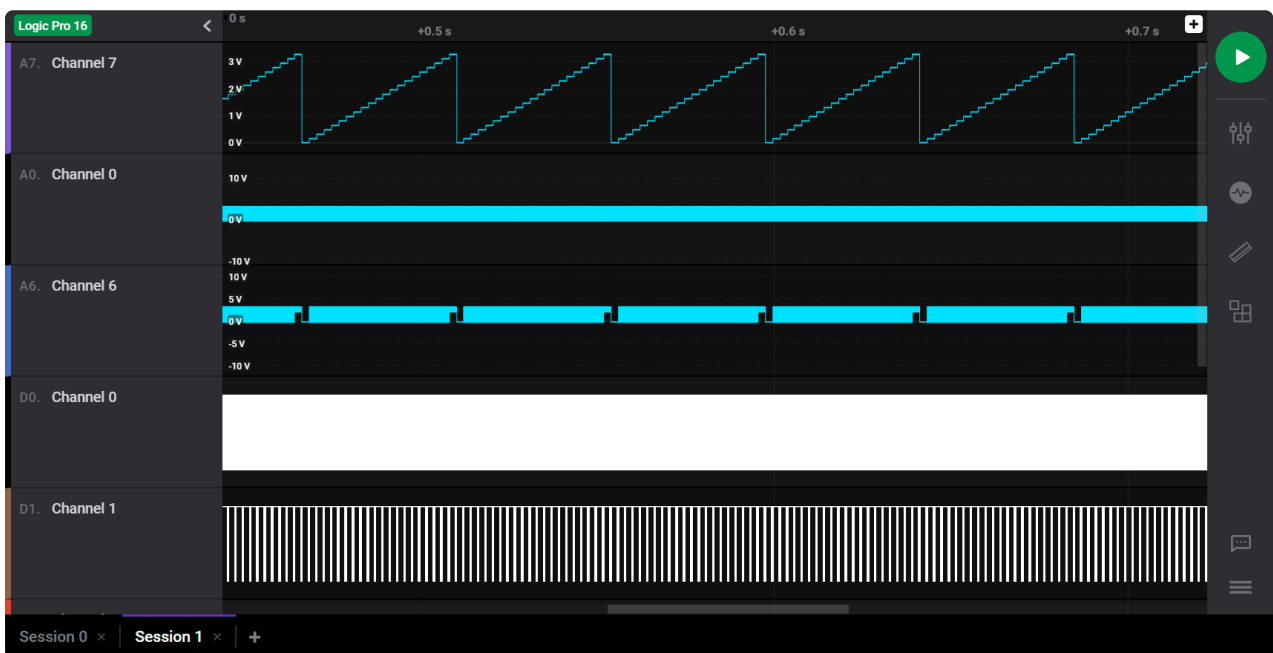
Customize your High Level Analyzer

To edit the Sample HLA (perhaps as a starting point to creating your own HLA), you can click the 'Local' button next to 'Sample HLA' under the Extensions panel. This will open the containing folder for your extension files which you can update for your needs.

Check out our [High Level Analyzer](#) article to learn more about customizing your HLA.

Using a Measurement Extension

The software currently has a few built-in measurements already installed and ready to use. The gif below demonstrates how to use them. You can also hold the shift key while dragging across your recorded data to add a measurement without using the sidebar. The use of measurement extensions allows additional custom measurements to be made.



Logic 2 measurements

To see your new measurement in action, take a capture of digital data and add a measurement to it as shown above. You should see the new measurements:



Measurement metrics

Customize your Measurement

To edit the Sample Measurement, you can click the 'Local' button next to 'Sample Measurement' under the Extensions panel. This will open the containing folder for your extension files which you can update for your needs.

Check out our [Measurement](#) article to learn more about customizing your Measurement.

High-Level Analyzer (HLA) Extensions

Learn how to modify your new High Level Analyzer

This guide assumes that you have familiarity with the [Python](#) programming language. It is what will be used to customize our HLA.

Overview

This guide assumes [you have generated](#) a new High-Level Analyzer. In this guide you will learn about:

1. The files included in the HLA template extension and what they are.
2. The different parts of `HighLevelAnalyzer.py`.
3. How to process input analyzer frames and output new analyzer frames.

High Level Analyzer Files

In your new High Level Analyzer (HLA) extension folder you will find 3 files:

- `README.md`
 - Documentation for your extension, shown within Logic 2 when you select an extension, and what users will see if you put your extension on the Marketplace.
- `extension.json`
 - Every extension must have this file in its root directory.
 - Contains metadata about the extension, and the HLAs and Measurement scripts that are included with the extension.
 - See [Extension File Format](#) for more information.
- `HighLevelAnalyzer.py`
 - Python source code for your HLA.

For the purposes of this document, we will be focusing on `HighLevelAnalyzer.py`

HighLevelAnalyzer.py Breakdown

Let's break down the contents of `HighLevelAnalyzer.py` .

HighLevelAnalyzer.py

```
from saleae.analyzers import HighLevelAnalyzer, AnalyzerFrame, StringSetting

class MyHla(HighLevelAnalyzer):

    # Settings:
    my_string_setting = StringSetting()
    my_number_setting = NumberSetting(min_value=0, max_value=100)
    my_choices_setting = ChoicesSetting(['A', 'B'])

    # Output formats
    result_types = {
        'mytype': {
            'format': 'Output type: {{type}}, Input type: {{data.input_type}}
        }
    }

    # Initialization
    def __init__(self):
        print("Settings:", self.my_string_setting,
              self.my_number_setting, self.my_choices_setting)

    # Data Processing
    def decode(self, frame: AnalyzerFrame):
        return AnalyzerFrame('mytype', frame.start_time, frame.end_time,
                              'input_type': frame.type
        })
```

Imports

```
from saleae.analyzers import HighLevelAnalyzer, AnalyzerFrame, StringSetting
```

Declaration and Settings

All HLAs must subclass [HighLevelAnalyzer](#), and additionally output [AnalyzerFrames](#). The `Setting` classes are included so we can specify settings options within the Logic 2 UI.

```
class MyHla(HighLevelAnalyzer):
    my_string_setting = StringSetting(label='My String')
    my_number_setting = NumberSetting(label='My Number', min_value=0, max
    my_choices_setting = ChoicesSetting(label='My Choice', ['A', 'B'])
```

This declares our new HLA class, which extends from HighLevelAnalyzer, and 3 settings options that will be shown within the Logic 2 UI. Note: if the name of the class MyHla() is used it must be referenced in the accompanying json file. Note that Hla() is the default value.

Output formats

```
result_types = {
    'mytype': {
        'format': 'Output type: {{type}}', Input type: {{data.input_ty
    }
}
```

This specifies how we want output AnalyzerFrames to be displayed within the Logic 2 UI. We will come back to this later.

Initialization

```
def __init__(self):
    print("Settings:", self.my_string_setting,
          self.my_number_setting, self.my_choices_setting)
```

This is called when your HLA is first created, before processing begins. The values for the settings options declared at the top of this class will be available as instance variables here. In this case, the settings values will be printed out and visible within the Logic 2 terminal view.

Data Processing

```
def decode(self, frame: AnalyzerFrame):
    return AnalyzerFrame('mytype', frame.start_time, frame.end_time,
        'input_type': frame.type
    )
```

This is where the bulk of the work will be done. This function is called every time the input to this HLA produces a new frame. It is also where we can return and output new frames, to be displayed within the Logic 2 UI. In this case we are outputting a new frame of type `'mytype'`, which spans the same period of time as the input frame, and has 1 data value `'input_type'` that contains the value of the `type` of the input frame.

HLA Debugging Tips

Although we don't have the ability to attach debuggers to Python extensions at the moment, here are some suggestions to help debug your HLA.

- Use `print()` statements to print debug messages to our in-app terminal. More information on our in-app terminal can be found below.

PAGE

Data Table & Terminal View



- Use the Wall Clock Format and Timing Markers to locate the exact frame listed in your error message.

PAGE

Time Bar Settings

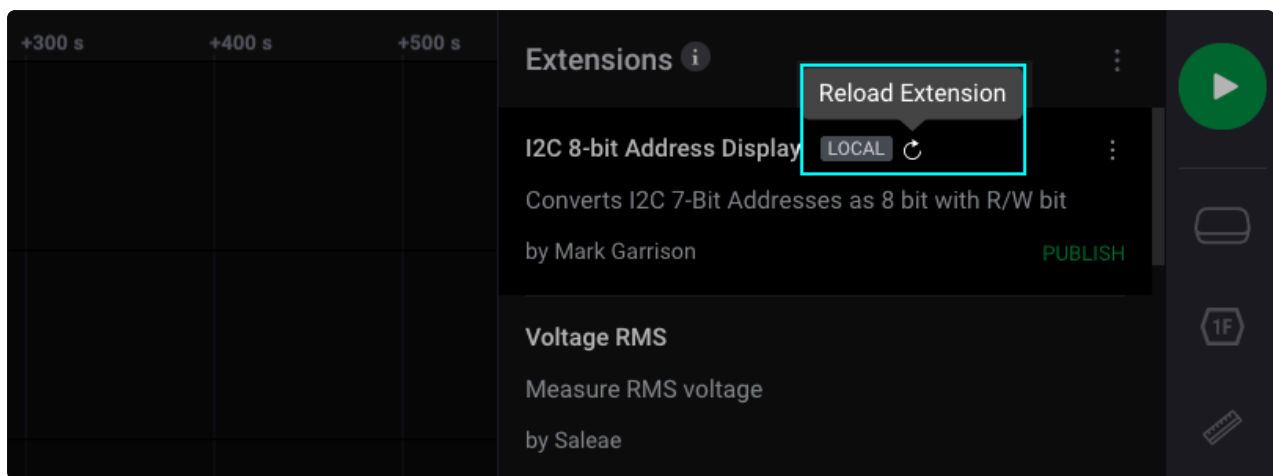


PAGE

Measurements, Timing Markers & Notes



- Use the reload source button in the app to quickly re-run your HLA after editing your source code.



"Reload Extension" button

Example - Writing an HLA to search for a value

Now that we've gone over the different parts of an HLA, we will be updating our example HLA to search for a value from an Async Serial analyzer.

Example Data

In the Extensions Quickstart you should have downloaded and opened a capture of i2c data. For this quickstart we will be using a capture of Async Serial data that repeats the message "Hello Saleae".



10KB

hla-quickstart.zip
archive

Remove Unneeded Code

To start, let's remove most of the code from the example HLA, and replace the settings with a single `search_for` setting, which we will be using later.

```

from saleae.analyzers import HighLevelAnalyzer, AnalyzerFrame, StringSett

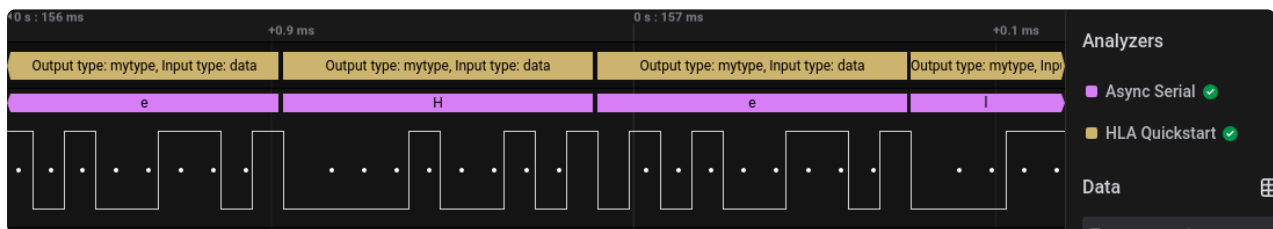
class MyHla(HighLevelAnalyzer):
    search_for = StringSetting()
    result_types = {
        'mytype': {
            'format': 'Output type: {{type}}, Input type: {{data.input_ty
        }
    }

    def __init__(self):
        pass

    def decode(self, frame: AnalyzerFrame):
        return AnalyzerFrame('mytype', frame.start_time, frame.end_time,
            'input_type': frame.type
        })

```

If you open the example data from above and add this analyzer, selecting the Async Serial analyzer as input, you should see the following when zooming in:



Our HLA (top) is outputting a frame for every frame from the input analyzer (bottom), and displaying their types.

Understanding the Input Frames

The goal is to search for a message within the input analyzer, but first we need to understand what frames the input analyzer (Async Serial in this case) produces so we can know what frames will be passed into the `decode(frame: AnalyzerFrame)` function.

The frame formats are documented under [Analyzer Frame Types](#), where we can find [Async Serial](#).

The Async Serial output is simple - it only outputs one frame type, `data`, with 3 fields: `data`, `error`, and `address`. The serial data we are looking at will not be configured to produce frames with the `address` field, so we can ignore that.

To recap, the `decode(frame)` function in our HLA will be called once for each frame from the Async Serial analyzer, where:

- `frame.type` will always be `data`
- `frame.data['data']` will be a ``bytes`` object with the data for that frame
- `frame.data['error']` will be set if there was an error

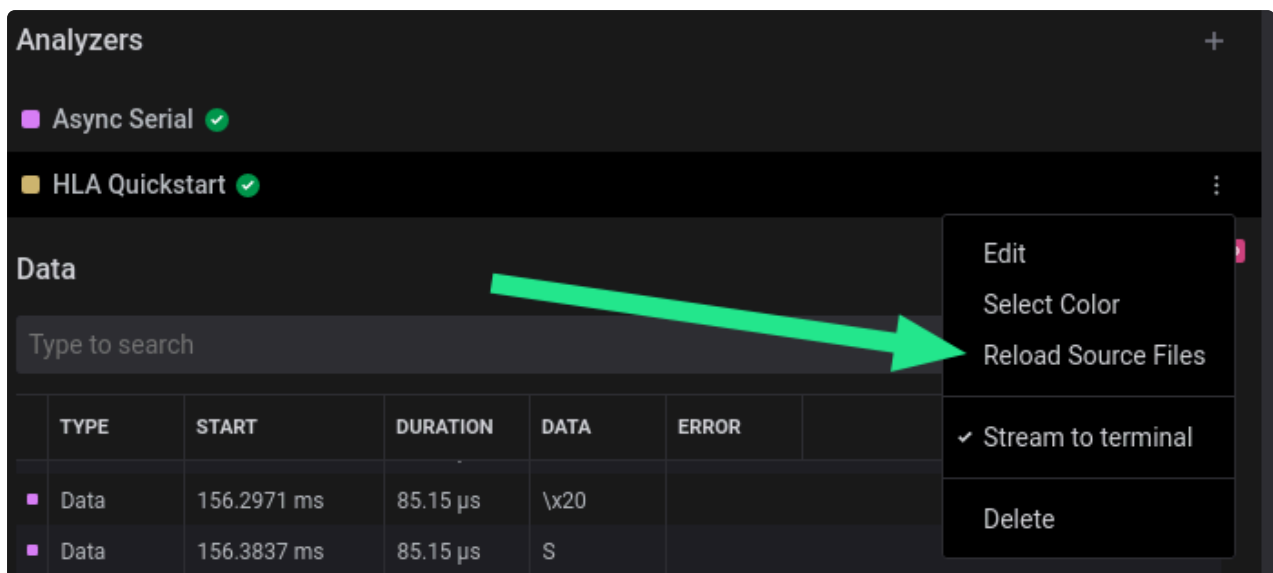
Updating `decode()` to search for "H" or "I"

Now that we understand the input data, let's update our HLA to search for the character `"H"`.

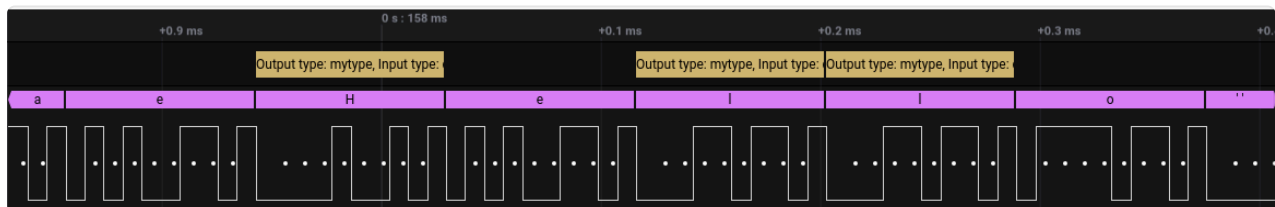
```
def decode(self, frame: AnalyzerFrame):
    # The `data` field only contains one byte
    try:
        ch = frame.data['data'].decode('ascii')
    except:
        # Not an ASCII character
        return

    # If ch is 'H' or 'I', output a frame
    if ch in 'HI':
        return AnalyzerFrame('mytype', frame.start_time, frame.end_time,
                             'input_type': frame.type
                             {})
```

After applying the changes, you can open the menu for your HLA and select `Reload Source Files` to reload your HLA:



You should now only see HLA frames where the Async Serial frame is an `H` or `l`:



Replace the hardcoded search with a setting

Now that we can search for characters, it would be much more flexible to allow the user to choose the characters to search for - this is where our `search_for` setting that we added earlier comes in.

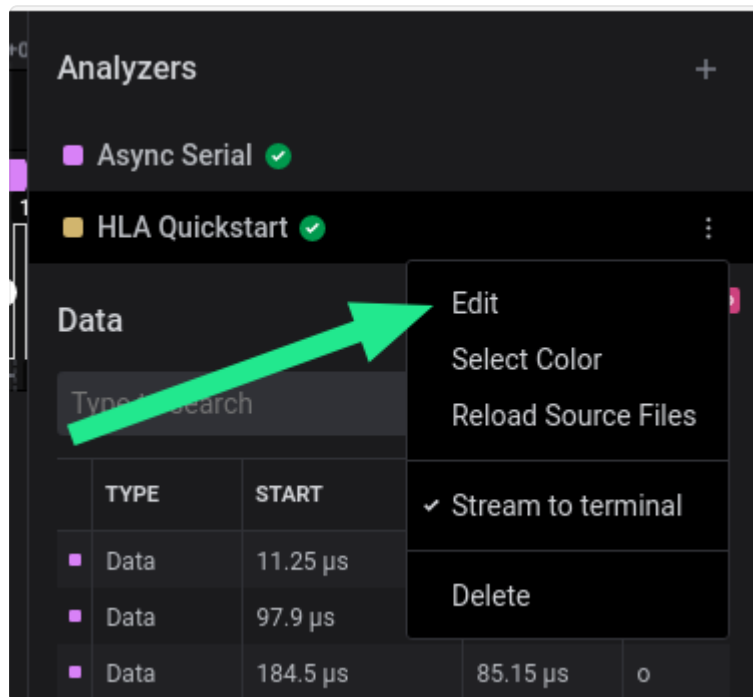
```
class MyHla(HighLevelAnalyzer):
    search_for = StringSetting()
```

Instead of using the hardcoded `'Hl'`, let's replace that with the value of `search_for`:

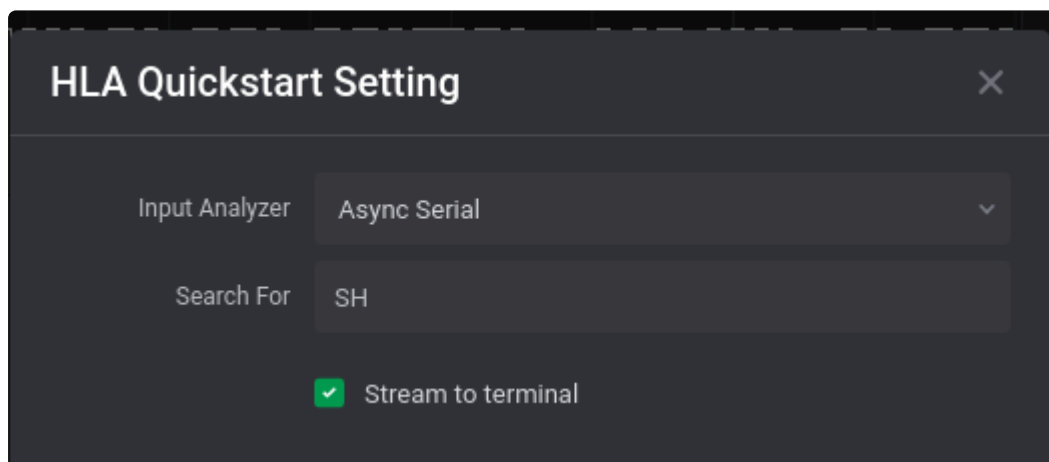
In `decode()`

```
# If the character matches the one we are searching for, output a new fra
if ch in self.search_for:
    return AnalyzerFrame('mytype', frame.start_time, frame.end_time, {
        'input_type': frame.type
    })
```

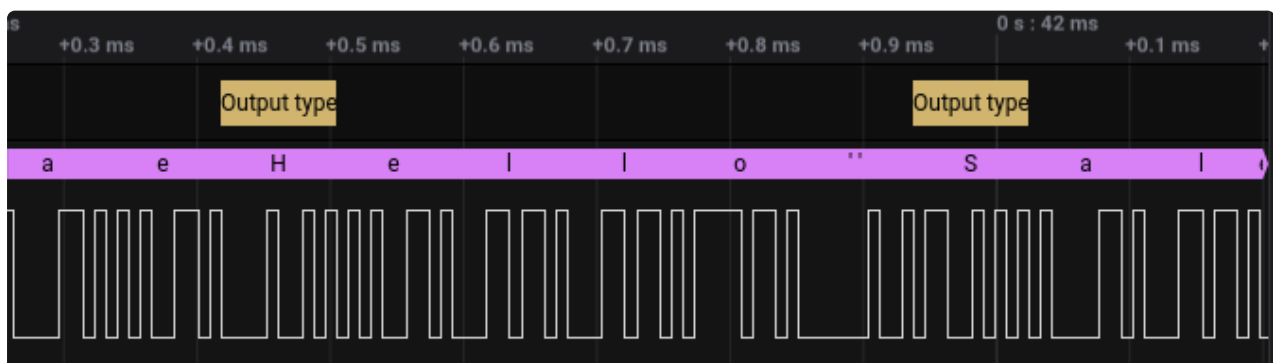
Now if you can specify the characters to search for in your HLA settings:



Click Edit to show the settings



Set the "Search For" setting



Now only the values 'S' and 'H' have frames

Updating the display string

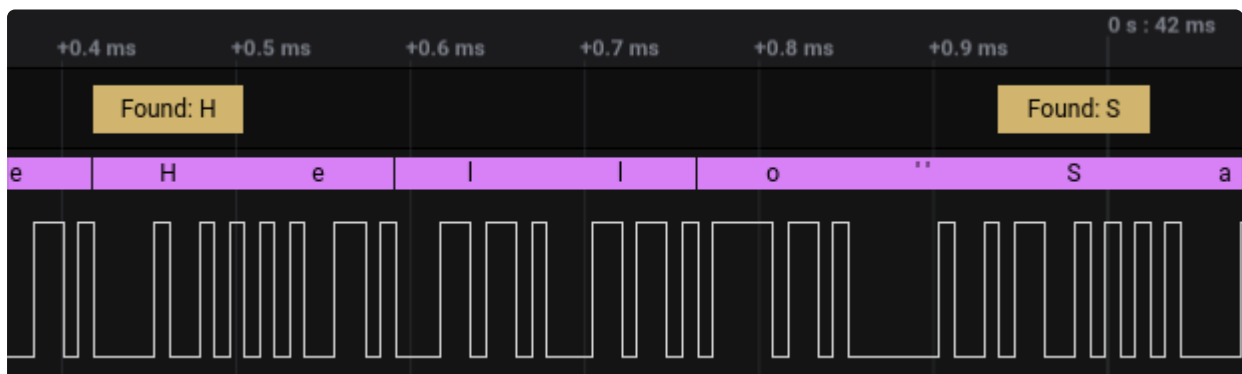
To update the display string shown in the analyzer bubbles, the `format` string in `result_types` variable will need to be updated. `'mytype'` will also be updated to `'match'` to better represent that the frame represents a matched character.

```
result_types = {
    'match': {
        'format': 'Found: {{data.char}}'
    }
}
```

And in `decode()`: we need to update the data in `AnalyzerFrame` to include `'char'`, and update the frame `'type'` to `'match'`.

```
# If the character matches the one we are searching for, output a new fra
if ch in self.search_for:
    return AnalyzerFrame('match', frame.start_time, frame.end_time, {
        'char': ch
    })
```

After reloading your HLA you should see the new display strings:



That's a lot more descriptive!

Using time

`AnalyzerFrame`s include a `start_time` and `end_time`. These get passed as the second and third parameter of `AnalyzerFrame`, and can be used to control the time span of a frame. Let's use it to fill in the gaps between the matching frames.

Let's add a `__init__()` to initialize the 2 time variables we will use to track the span of time that doesn't have a match:

```
def __init__(self):
    self.no_match_start_time = None
    self.no_match_end_time = None
```

And update `decode()` to track these variables:

```
# If the character matches the one we are searching for, output a new fra
if ch in self.search_for:
    frames = []

    # If we had a region of no matches, output a frame for it
    if self.no_match_start_time is not None and self.no_match_end_time is
        frames.append(AnalyzerFrame(
            'nomatch', self.no_match_start_time, self.no_match_end_time,

            # Reset match start/end variables
            self.no_match_start_time = None
            self.no_match_end_time = None

        frames.append(AnalyzerFrame('match', frame.start_time, frame.end_time
            'char': ch
        )))

    return frames
else:
    # This frame doesn't match, so let's track when it began, and when it
    if self.no_match_start_time is None:
        self.no_match_start_time = frame.start_time
    self.no_match_end_time = frame.end_time
```

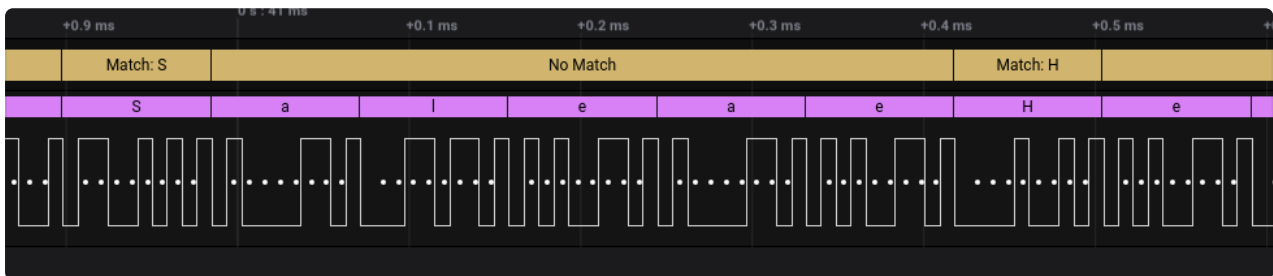
And lastly, add an entry in `result_types` for our new `AnalyzerFrame` type `'nomatch'` :

```

result_types = {
    'match': {
        'format': 'Match: {{data.char}}'
    },
    'nomatch': {
        'format': 'No Match'
    }
}

```

The final output after reloading:




What's Next?

- Find out about other analyzers and the AnalyzerFrames they output in the [Analyzer Frame Types](#) documentation.
- Use the [API Documentation](#) as a reference.
- Browse the Saleae Marketplace in Logic 2 for more ideas and examples of extensions you can create.
- [Publish your extension](#) to the Saleae Marketplace!

Measurement Extensions

Learn how to modify your new Measurement

 This guide assumes that you have familiarity with the [Python](#) programming language. It is what will be used to customize our Measurement.

Overview

This guide assumes [you have generated](#) a new Digital Measurement. In this guide you will learn about:

1. The files included in the Digital Measurement template extension and what they are.
2. The different parts of `DigitalMeasurement.py`.
3. How to update the template to make your own measurements.

Measurement Files

In your new Measurement extension folder you will find 3 files:

- `README.md`
 - Documentation for your extension, shown within Logic 2 when you select an extension, and what users will see if you put your extension on the Marketplace.
- `extension.json`
 - Every extension must have this file in its root directory.
 - Contains metadata about the extension, and the HLAs and Measurement scripts that are included with the extension.
 - See [Extension File Format](#) for more information.
- `DigitalMeasurement.py` or `AnalogMeasurement.py`
 - Python source code for your measurement.

DigitalMeasurement.py and AnalogMeasurement.py

Digital measurements are implemented with a class template that looks like below. Take a look at `pulseCount.py` to see how this was modified for our Pulse Count extension.

```
from saleae.range_measurements import DigitalMeasurer

class MyDigitalMeasurement(DigitalMeasurer):
    # Add supported_measurements here. This includes the metric
    # strings that were defined in the extension.json file.

    def __init__(self, requested_measurements):
        super().__init__(requested_measurements)
        # Initialize your variables here

    def process_data(self, data):
        for t, bitstate in data:
            # Process data here

    def measure(self):
        values = {}
        # Assign the final metric results here to the values object
        return values
```

Analog measurements are implemented with a class template that looks like below. Take a look at `voltage_peak_to_peak.py` to see how this was modified for an example analog extension.


```

from saleae.range_measurements import AnalogMeasurer

class VoltageStatisticsMeasurer(AnalogMeasurer):
    # Add supported_measurements here. This includes the metric
    # strings that were defined in the extension.json file.

    def __init__(self, requested_measurements):
        super().__init__(requested_measurements)
        # Initialize your variables here

    def process_data(self, data):
        # Process data here

    def measure(self):
        values = {}
        # Assign the final metric results here to the values object
        return values

```

Measurer creation

In python, your class will be constructed when the user adds or edits a measurement. This instance of your class will be used for a single computation. Your class can either process analog data or digital data, but not both. A class may handle as many metrics as you want though. If you want to implement both digital and analog measurements, you will need at a minimum two classes.

Constructor

The constructor will be called with an array of requested measurements, which are taken from the `extension.json` file. In our `pulseCount.py` example, this is declared like so:

```

POSITIVE_PULSES = 'positivePulses'
NEGATIVE_PULSES = 'negativePulses'

class PosNegPulseMeasurer(DigitalMeasurer):
    supported_measurements = [POSITIVE_PULSES, NEGATIVE_PULSES]

```

Process Data

Immediately after construction, the function `def process_data(self, data):` will be called one or more times. This function takes a parameter `data` which differs between analog and digital measurements.

The Saleae Logic software stores collected data in chunks. To keep python processing performant, the Logic software passes these blocks, or sections of these blocks, one at a time to your measurement. If the requested measurement range does not line up with the internal block storage, the objects passed to python will already be adjusted to the measurement range, so no work needs to be done to handle this condition.

This makes it impossible to know exactly how much data will be needed for the given measurement range the first time `process_data` is called. Be sure to update the internal state of your class in such a way that this isn't a problem. For example, when computing the average analog value over a range, it would be best to hold the sum of all values passed to `process_data` and the total count of samples in data members, and only compute the average in the `measure` function.

Process Analog Measurements

For analog measurements, `data` is an instance of `AnalogData`, which is an `iterable` class with the properties `sample_count` and `samples`. `sample_count` is a number, and is the number of analog samples in the data instance. Note - this might not be the total number of analog samples passed to your measurement, since `process_data` may be called more than once if the user selected range spans multiple chunks.

`samples` is a [numpy ndarray](#). For information handling this type, please refer to the [numpy documentation](#).

The `process_data` function should not return a value. Instead, it should update the internal state of your class, such that the `measure` function can produce your measurement's results.

Process Digital Measurements

For digital measurement classes, the `data` parameter is an instance of the `iterable` Saleae class `DigitalData`. Each iteration returns a pair of values - the current time, as a `GraphTime` class instance, and the current bit state as a `bool`. (`True` = signal high).

The object is essentially a list with the timestamp of each transition inside of the user selected region of digital data.

The `GraphTime` has one feature. One `GraphTime` can be subtracted from another to compute the difference in seconds, as a `GraphTimeDelta`. `GraphTimeDelta` can be converted to a float using `float(graph_time_delta)`. This allows your code to compute the time in between transitions, or the time duration between the beginning of the measurement and any transition inside of the measurement range, but it does not expose absolute timestamps.

For example, to compute the total number of transitions over the user selected range, this could be used:

```
def __init__(self, requested_measurements):
    super().__init__(requested_measurements)
    self.first_transition_time = None
    self.edge_count = 0

def process_data(self, data):
    for t, bitstate in data:
        if self.first_transition_time is None:
            self.first_transition_time = t
        else:
            # note: the first entry does not indicate a transition, it's si
            self.edge_count += 1
```

Currently, the `DigitalData` collection will first include the starting time and bit state, and then every transition that exists in the user selected range, if any. It also exposes the starting and ending time of the user-selected measurement range. Consult the [API Documentation](#) for details.

Measure

The `def measure(self):` function will be called on your class once all data has been passed to `process_data`. This will only be called once and should return a dictionary with one key for every `requested_measurements` entry that was passed into your class's constructor.

i in the future, we may allow the user to select which metrics to compute. To avoid unnecessary processing, it's recommended to check the `requested_measurements` provided by the constructor before computing or returning those values. However, returning measurements that were not requested is allowed. The results will just be ignored.

What's Next?

- Browse the Saleae Marketplace in Logic 2 for more ideas and examples of extensions you can create.
- [Publish your extension](#) to the Saleae Marketplace!

PAGE

Publish an Extension



Extension File Format

Extensions are composed of at least three files, `extension.json`, `readme.md`, and one or more python files. The Logic 2 software uses **Python version 3.8**.

extension.json File Layout

A single extension can contain multiple high level analyzers, measurements, or both.

This example is a for a single extension that contains one high level analyzer and one digital measurement.

```
{
  "version": "0.0.1",
  "apiVersion": "1.0.0",
  "author": "Mark Garrison",
  "description": "Utilities to measure I2C speed and utilization",
  "name": "I2C Utilities",
  "extensions": {
    "I2C EEPROM Reader": {
      "type": "HighLevelAnalyzer",
      "entryPoint": "I2cUtilities.Eeprom"
    },
    "I2C clock speed measurement": {
      "type": "DigitalMeasurement",
      "entryPoint": "I2cUtilities.I2cClock",
      "metrics": {
        "i2cClockSpeed": {
          "name": "I2C Clock Speed",
          "notation": "N MHz"
        },
        "i2cBusUtilization": {
          "name": "I2C Bus Utilization",
          "notation": "N %"
        }
      }
    }
  }
}
```

The `author`, `description`, and `name` properties manage what appears in the Extensions panel where the extensions are managed.

Property	Example	Description
version	"0.0.1"	The version if this extension. Increase this when publishing updates in order to allow users to update.
apiVersion	"1.0.0"	The Saleae API version used. This should remain "1.0.0" until we make changes to the extension API.
author	"Mark Garrison"	Your name
description	"Utilities to measure I2C..."	A one line description for display in the marketplace
name	"I2C Utilities"	The name of the extension package, for display. Note - this does not have to match the names of the individual extensions, which are shown elsewhere.
extensions	see below	An object with one key per high level analyzer or measurement.

The `extensions` section describes each high level analyzer or measurement that is contained in this extension package.

In the example above, there are 2 extensions listed.

```

"extensions": {
  "I2C EEPROM Reader": {
    "type": "HighLevelAnalyzer",
    "entryPoint": "I2cUtilities.Eeprom"
  },
  "I2C clock speed measurement": {
    "type": "DigitalMeasurement",
    "entryPoint": "I2cUtilities.I2cClock",
    "metrics": {
      "i2cClockSpeed": {
        "name": "I2C Clock Speed",
        "notation": "N MHz"
      },
      "i2cBusUtilization": {
        "name": "I2C Bus Utilization",
        "notation": "N %"
      }
    }
  }
}

```

There are three different types of extension here, as indicated by the type property. "HighLevelAnalyzer", "DigitalMeasurement", and "AnalogMeasurement". There are some differences between these types.

Type: HighLevelAnalyzer

Property	Example	Description
type	"HighLevelAnalyzer"	This must be the string "HighLevelAnalyzer"
entryPoint	"I2cUtilities.Eeprom"	The python entry point. The first part is the python filename without the extension, and the second part is the name of the class in that file

Type: AnalogMeasurement and DigitalMeasurement

Property	Example	Description
----------	---------	-------------

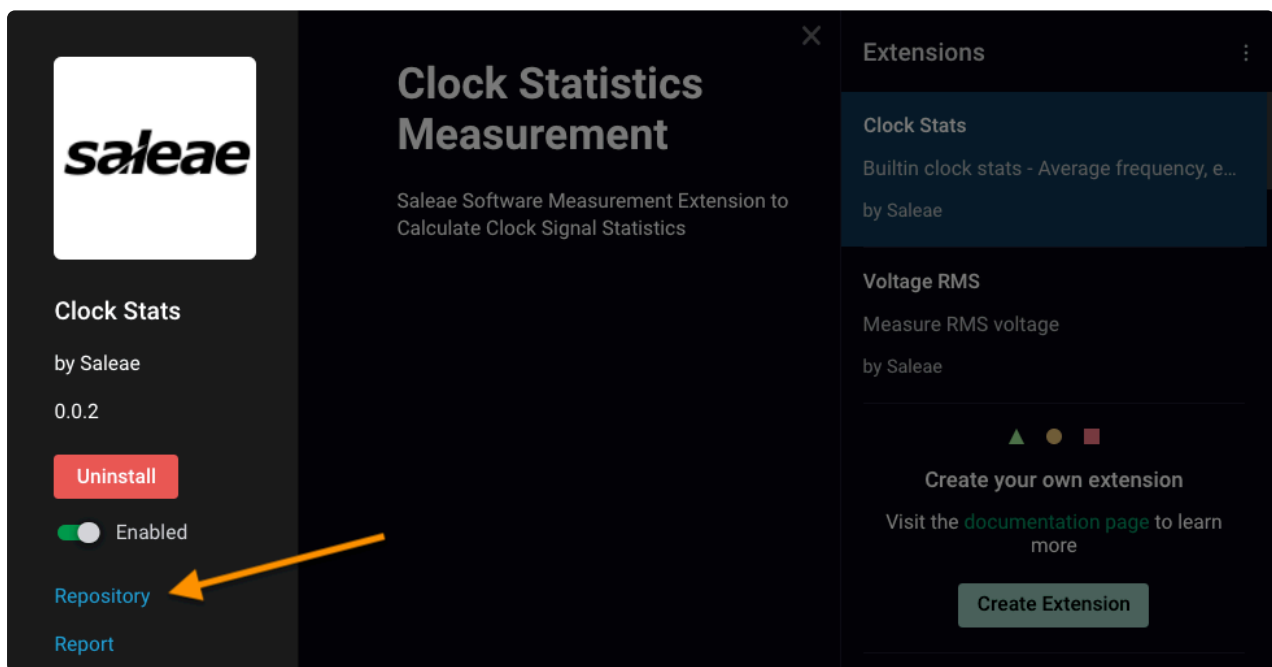
type	"DigitalMeasurement"	This must be either "DigitalMeasurement" or "AnalogMeasurement"
entryPoint	"I2cUtilities.I2cClock"	The python entry point. The first part is the python filename without the extension, and the second part is the name of the class in that file
metrics	"i2cClockSpeed": { "name": "I2C Clock Speed", "notation": "N MHz" }	Object that contains one property for each single numeric metric you wish to display. If your measurement class produces 4 different numbers, all 4 must be present here. Each metric property name must match the name of the metric produced by your python class.
metrics[X].name	"I2C Clock Speed"	The display name for a given metric
metrics[X].notation	"N MHz"	The display string for formatting the metric. Limited HTML tags are supported: <div>['b', 'i', 'em', 'strong', 'sub']</div>

About Third-Party Extensions

Our Extensions Marketplace allows you to publish extensions that you create, or to install extensions published by other members of the community. Extensions are community-written Python modules. Extensions are not installed automatically, except for a few Saleae-created extensions.

Before installing an extension, please keep in mind the following:

- Non-Saleae extensions were created and published by members of the community.
- Saleae cannot endorse their contents.
- Please treat extensions the same way you would other programs downloaded from the internet.
- All extensions are open source and can viewed by clicking the "Repository" link in the extension details.



Repository link from within the extension details

If you have any problems with an extension, please be sure to create an issue on the extension's GitHub repository. If you suspect an extension does not comply with the [Saleae Marketplace Partner Agreement](#), please [contact Saleae support](#) or click on the "Report" button from within the extension details.

Publish an Extension

Publishing your extension to the Saleae Marketplace will make it readily available to anyone who uses our software. With your help, we're hoping to provide a growing list of feature extensions that our users can benefit from!

Prerequisites

Before publishing your extension, you will need to have the following completed.

- You have finished developing your extension and are ready to share it. You can follow along with our Extensions Quickstart guide below as a starting point.

PAGE

Create and Use Extensions

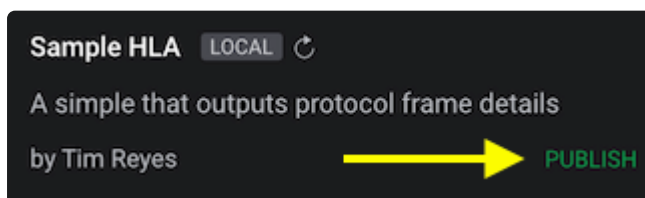
>

- Your extension must be uploaded to a [GitHub](#) repository.
- You must [create a release](#) for your extension.

You can take a look at an [example GitHub repository](#) for our Sample HLA, which we will use in the following guide.

Publishing an Extension

1. Once you've finished developing your extension, click 'Publish' under the Extensions panel for your extension.



2. Clicking 'Publish' should open your browser and load our extensions submission page. Provide your GitHub repository URL here and click 'Submit'.



Submit the GitHub link of your extension, then approve Saleae Marketplace

Submit

- i** If clicking the Publish button doesn't open your browser, you can manually reach our Extensions Marketplace Publish site via the link below:

<https://marketplace.saleae.com/publish>

3. Afterwards, you'll be taken to a new page to authorize Saleae Marketplace to access your GitHub account.

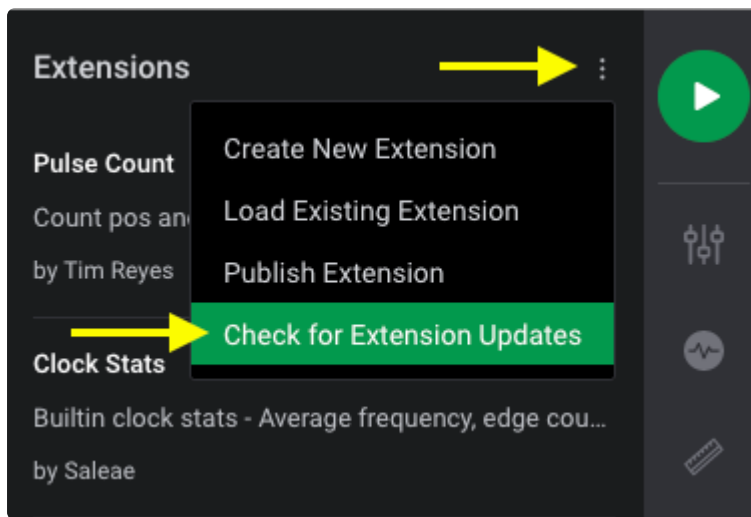
- i** If clicking "Submit" takes you to an error page on GitHub's website, please ensure you are logged in with an account that has administrator permissions over the repository you are attempting to publish. Also, if the repository is part of an organization, there may be some policies for the organization that might be causing the error.

4. Click 'Authorize' and you should immediately receive an email confirming that your extension has been added to the Marketplace. The web page should also confirm that your extension was submitted successfully.

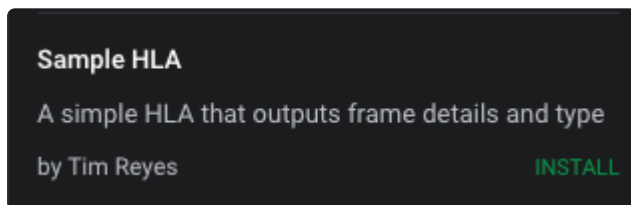


Extension submitted successfully!

5. Click the three dots at the top of the Extensions panel and click 'Check for Extension Updates.'



6. Congratulations! If the publish was successful, your extension should now appear in the software like below.



HLA - Analyzer Frame Format

Documentation for output produced by Saleae's built-in Analyzers

Python High Level Analyzers allow users to write custom code that processes the output of an analyzer. The below list of pre-installed low level analyzers can be immediately used with high level analyzers (HLAs).

- [Async Serial](#)
- [I2C](#)
- [SPI](#)
- [CAN](#)
- [LIN](#)
- [Manchester](#)
- [Parallel](#)
- [LED](#)
- [1-Wire](#)
- [I2S / PCM](#)

Adding HLA Support for More Analyzers

We've released documentation on our FrameV2 API below, which can be used to add HLA support for any low level analyzer that is not listed above, including custom analyzers that were created using our Protocol Analyzer SDK.

PAGE

FrameV2 / HLA Support - Analyzer SDK



Writing an HLA

In order to write a high level analyzer, the data format produced by the connected source analyzer must be understood.

For example, a high level analyzer which consumes serial data needs to understand the serial analyzer output format in order to extract bytes, and this code is very different from the code required to extract data bytes from CAN data.

```
# Reading serial data

def decode(self, frame):
    print(frame.data['data'])

# Reading CAN data

def decode(self, frame):
    if frame.type == 'identifier_field':
        print(frame.data['identifier'])
    elif frame.type == 'data_field':
        print(frame.data['data'])
    elif frame.type == 'crc_field':
        print(frame.data['crc'])
```

To write a Python high level analyzer for a specific input analyzer, navigate to the section for that analyzer.

There is one page for each analyzer that is compatible with python HLAs. Each analyzer produces one or more frame types. These frame types have string names and will be in the `type` member of the frame class.

Each frame type may have data properties. The documentation page will list the data properties for that frame type, along with the data type and a description. These can be accessed from the `frame.data` dictionary.

Here is an example of you you might handle different frame types from I2C:

```

def decode(self, frame):
    if frame.type == 'address':
        if frame.data['read'] == True:
            print('read from ' + str(frame.data['address'][0]))
        else:
            print('write to ' + str(frame.data['address'][0]))
    elif frame.type == 'data':
        print(frame.data['data'][0])
    elif frame.type == 'start':
        print('I2C start condition')
    elif frame.type == 'stop':
        print('I2C stop condition')

```

State Managment

In some cases, such in I2C HLA programming, each frame only contains a single byte. Therefore a state machine is needed to keep track of the bytes as they are received so that they can be properly interpreted, specifically if multiple control modes or multi-byte instructions are required. For example the first byte of the data payload is often the slave address, followed by a control byte. The control byte would determine if follow-on bytes should be interpreted as commands or memory register data. The following example provides some recommendations for state management.

An example set of states follows:

1. *Idle*: Waiting for I2C transaction to begin
 - Next State: waiting for Start
2. *Start*: I2C start condition signals the beginning of the transaction
 - Next State: waiting for the expected slave device Address
3. *I2C Slave Address*: Typically a 7-bit device address is read, direction bit (read/write) is determined
 - Next State: waiting for waiting for control byte
 - Note that the I2C LLA automatically shifts the address back right by 1 bit to recover the original 7-bit address.
4. *I2C Control Byte*: When used, it often determines how follow-on bytes should be interpreted (commands or data)
 -

- Next State: waiting for data
5. *Data*: Data can be supplied as one or more bytes
 - The I2C LLA (low level analyzer) pass data bytes, one byte at a time to the HLA for decoding.
 - If a multi-byte instruction is received, tracking of the previous byte(s) may be required to interpret the current byte correctly.
 - Next State: Loop to receive next Data byte until Stop is received
 6. *Stop*: I2C stop condition signals the end of the transaction
 - Next State: return to Idle

Instruction Set Lookup Table

Developing an instruction set lookup table in JSON is best practice that allows you to quickly build an analyzer that interprets received data into human readable annotations. Often at a minum the instruction, name, and numbrer of parameters is needed to build out the state machine.

```
instructions = {
  0x81: {"name": "Set Contrast Control", "param_description": "Contrast
  0xA4: {"name": "Entire Display OFF", "param_description": "", "params
  0xA5: {"name": "Entire Display ON", "param_description": "", "params"
}
```


Async Serial - Frame Format

Output Frame Format

Frame Type: "data"

- A single serial word

Property	Type	Description
<code>data</code>	bytes	The serial word, the width in bits is controlled by the serial settings
<code>error</code>	str	(optional) Present if an error was detected when decoding this word
<code>address</code>	bool	(optional) Present if multi-processor or multi-drop bus special modes were selected. True indicates that this is an address byte

I2C - Frame Format

I2C Communication Overview

I2C communication involves two roles: a master and a slave. The master initiates and controls the communication, while the slave responds to the master's requests. Data transmission on the I2C bus is organized into transactions, each starting with a START condition and ending with a STOP condition. Each transaction begins with a 7-bit address and comprises one or more data bytes, with each byte followed by an acknowledgment from the receiver.

I2C Frame Evaluation

I2C communication is captured as frames by a built-in low-level I2C analyzer. Each frame represents one byte of information. Frame types are defined below including: start, address, data, and stop.

Data Frame Evaluation

The HLA executes once for each incoming frame. Each frame contains only one byte. Each transaction begins with a start frame, an address frame, and then any number of data frames, ending with a stop frame. A state machine is often needed to decode multi-byte (multi-frame) sequences such as a command followed by a parameter.

Output Frame Format

Frame Type: "address"

- I2C address byte

Property	Type	Description
address	byte	The 7 bit I2C address
read	bool	True for read operations, false for write operations
ack	bool	True when the address was ACKed, false when NAKed
error	str	(optional) Present if an there was a problem reading the I2C data

Example

Store the address as an integer and display it on the terminal as a hex value

```
if frame.type == 'address':
    address = frame.data['address'][0]
    print(f"Address: {hex(address)}")
```

Result

Data: 0x3C

Frame Type: "data"

- I2C data byte

Property	Type	Description
data	byte	single 8 bit data word
ack	bool	True when the data byte was ACKed, false when NAKed
error	str	(optional) Present if an there was a problem reading the I2C data

Example

Store the data value as an integer and display it on the terminal as a hex value

```
if frame.type == 'data':  
    data = frame.data['data'][0]  
    print(f>Data: {hex(data)}")
```

Result:

Data: 0x00

Frame Type: "start"

- undefined
- I2C start condition

Frame Type: "stop"

- undefined
- I2C stop condition

SPI - Frame Format

Output Frame Format

Frame Type: "enable"

Property	Type	Description
<ul style="list-style-type: none">• undefined• Indicates the enable (chip select) signal has transitioned from inactive to active, present when the enable channel is used		

Frame Type: "disable"

Property	Type	Description
<ul style="list-style-type: none">• undefined• Indicates the enable signal has transitioned back to inactive, present when the enable channel is used		

Frame Type: "result"

Property	Type	Description
<code>miso</code>	bytes	Master in slave out, width in bits is determined by settings
<code>mosi</code>	bytes	Master out slave in, width in bits is determined by settings

- A single word transaction, containing both MISO and MOSI

Frame Type: "error"

Property	Type	Description
<ul style="list-style-type: none">undefinedIndicates that the clock was in the wrong state when the enable signal transitioned to active		

CAN - Frame Format

Output Frame Format

Frame Type: "identifier_field"

Property	Type	Description
<code>identifier</code>	int	Identifier, either 11 bit or 29 bit
<code>extended</code>	bool	(optional) Indicates that this identifier is a 29 bit extended identifier. This key is not present on regular 11 bit identifiers
<code>remote_frame</code>	bool	(optional) Present and true for remote frames

Frame Type: "control_field"

Property	Type	Description
<code>num_data_bytes</code>	int	Number of data bytes in the transaction

Frame Type: "data_field"

Property	Type	Description
<code>data</code>	int	The byte

Frame Type: "crc_field"

Property	Type	Description
crc	int	16 bit CRC value

Frame Type: "ack_field"

Property	Type	Description
ack	bool	True when an ACK was present

Frame Type: "can_error"

Property	Type	Description
----------	------	-------------

- undefined
- Invalid CAN data was encountered

Manchester - Frame Format

Output Frame Format

Frame Type: "data"

Property	Type	Description
data	int	The size in bytes is determined by the settings

- This is the decoded manchester word

1-Wire - Frame Format

Output Frame Format

Frame Type: "reset"

Property	Type	Description
----------	------	-------------

- Reset pulse

Frame Type: "presence"

Property	Type	Description
----------	------	-------------

- Presence Pulse

Frame Type: "rom_command"

Property	Type	Description
description	str	read, skip, search, or match
rom_command	bytes	The command byte

- ROM command. This is the first command issued by the master after a presence pulse

Frame Type: "crc"

Property	Type	Description
crc	bytes	The CRC byte

- 8 bit CRC, last part of the 64 bit identifier

Frame Type: "family_code"

Property	Type	Description
family	bytes	The family code, which is the first part of the 64 bit identifier

- The family code of the device ID

Frame Type: "id"

Property	Type	Description
id	int	48 bit integer, taken from the center of the 64 bit identifier

- The 48 bit device identifier

Frame Type: "data"

Property	Type	Description
data	bytes	A single data byte

- Data byte after the ROM command and identifier

Frame Type: "invalid_rom_command"

Property	Type	Description
rom_command	bytes	The ROM command byte

- Unknown ROM command

Frame Type: "alarm"

Property	Type	Description
<code>rom_command</code>	bytes	The command byte

- Alarm search command

Async RGB LED - Frame Format

Output Frame Format

Frame Type: "pixel"

Property	Type	Description
index	int	The index along the LED strip. Index 0 is the first LED
red	int	The red channel, [0-255]
green	int	The green channel, [0-255]
blue	int	The blue channel, [0-255]

- Represents a single RGB pixel value

Simple Parallel - Frame Format

Output Frame Format

Frame Type: "data"

Property	Type	Description
data	int	Data word, the width in bits is determined by the number of enabled data channels

- A single parallel word

LIN - Frame Format

Output Frame Format

Frame Type: "no_frame"

Property	Type	Description
		<ul style="list-style-type: none">• undefined• Inter-byte space

Frame Type: "header_break"

Property	Type	Description
		<ul style="list-style-type: none">• undefined• Header break

Frame Type: "header_sync"

Property	Type	Description
		<ul style="list-style-type: none">• undefined• Header sync

Frame Type: "header_pid"

Property	Type	Description

protected_id	int	6 bit protected Id
--------------	-----	--------------------

- Protected identifier

Frame Type: "data"

Property	Type	Description
data	int	Data byte
index	int	Index, 0-8, of the data byte inside of the transaction

Frame Type: "checksum"

Property	Type	Description
checksum	int	LIN checksum

- Checksum byte

Frame Type: "data_or_checksum"

Property	Type	Description
checksum	int	LIN checksum
data	int	Data byte
index	int	Index, 0-8, of the data byte inside of the transaction

- Unable to determine if this byte is a data byte or a checksum. It is technically valid as both. This occurs if a data byte, at index N, is equal to what the CRC should be if the transaction is N-1 bytes.

I2S - Frame Format

Output Frame Format

Frame Type: "error"

Property	Type	Description
<code>error</code>	str	Error details. I2C errors usually indicate the wrong number of bits inside of a frame

- I2S decode error

Frame Type: "data"

Property	Type	Description
<code>channel</code>	int	channel index. 0 or 1
<code>data</code>	int	Audio value. signed or unsigned, based on I2S/PCM analyzer settings

- A single sample from a single channel

API Documentation

`saleae.data`

***class* saleae.data.GraphTime(datetime: datetime, millisecond=0, *, microsecond=0, nanosecond=0, picosecond=0)**

A high-precision wall clock time.

The primary way to use this type is to subtract two `GraphTime` s to produce a `GraphTimeDelta` . `GraphTimeDelta` s may be freely added and subtracted from each other, and converted to floating point seconds. They can also be added to or subtracted from `GraphTime` s to produce a suitable offset `GraphTime` .

Constructs a `GraphTime` using a `datetime` , and optionally sub-millisecond precision values.

The sub-millisecond precision values must be convertible to float.

`__add__()`

Add a `GraphTimeDelta` value to produce a new `GraphTime` .

`__str__()`

Converts `GraphTime` to an ISO 8601 string with picosecond precision.

Timezone is always UTC, using the Z suffix.

`__sub__()`

Subtract a `GraphTime` or `GraphTimeDelta` value.

When subtracting a `GraphTime`, produces a `GraphTimeDelta`. When subtracting a `GraphTimeDelta`, produces a `GraphTime`.

`as_datetime(self: GraphTime)`

Produces a `datetime` value that is as close as possible to the given value.

The produced `datetime` is always timezone aware and in the UTC timezone. Local time can be procured using the standard Python `datetime` conversion functions.

`class saleae.data.GraphTimeDelta(second=0, millisecond=0, *, microsecond=0, nanosecond=0, picosecond=0)`

A high-precision duration.

Constructs a `GraphTimeDelta` using numerical values.

All values must be convertible to float. Multiple prefixes may be specified, the resulting value will be all the values added together.

`__add__()`

Add a `GraphTimeDelta` value to produce a new `GraphTimeDelta`.

`__eq__()`

Determine if two `GraphTimeDelta` values are equal, up to a tolerance.

`__float__()`

Convert to a floating point number of seconds. Note that this can cause a loss of precision for values > 1ms.

`__ge__()`

Determine if the first `GraphTimeDelta` value is greater than or equal to the second, up to a tolerance.

`__gt__()`

Determine if the first `GraphTimeDelta` value is greater than the second, up to a tolerance.

`__le__()`

Determine if the first `GraphTimeDelta` value is less than or equal to the second, up to a tolerance.

`__lt__()`

Determine if the first `GraphTimeDelta` value is less than the second, up to a tolerance.

`__ne__()`

Determine if two `GraphTimeDelta` values are not equal, up to a tolerance.

`__sub__()`

Subtract a `GraphTimeDelta` value to produce a new `GraphTimeDelta`.

`class saleae.data.AnalogData(raw_samples: ndarray, voltage_transform_gain: float, voltage_transform_offset: float, start_time: saleae.data.timing.GraphTime, end_time: saleae.data.timing.GraphTime)`

`iter()`

Iterates over the samples in this instance as voltage values.

sample_count

The number of samples contained in this instance.

slice_samples(slice: slice) -> saleae.data.AnalogData

Allows creating an AnalogData from a subset of this one's samples.

samples

Samples after applying voltage scaling.

saleae.range_measurements

class

saleae.range_measurements.DigitalMeasurer(requested_measurements: List[[str](#)])

***class* saleae.range_measurements.AnalogMeasurer(*args, **kwargs)**

saleae.analyzers

***class* saleae.analyzers.HighLevelAnalyzer()**

Base class for High Level Analyzers. Subclasses must implement the `decode()` function

decode(frame: saleae.analyzers.high_level_analyzer.AnalyzerFrame)

Decode a frame from an input analyzer, and return None or 1 or more `AnalyzerFrame` objects.

***class* saleae.analyzers.AnalyzerFrame**(type: [str](#), start_time: `saleae.data.timing.GraphTime`, end_time: `saleae.data.timing.GraphTime`, data: [dict](#) = None)

A frame produced by an analyzer. The types of frames and the fields in each will depend on the analyzer.

- **Variables**
- [type](#) ([str](#)) – Frame type
- **start_time** (`saleae.data.GraphTime`) – Start of frame
- **end_time** (`saleae.data.GraphTime`) – End of frame
- **data** ([dict](#)) – Key/value data associated with this frame

***class* saleae.analyzers.StringSetting**(**kwargs)

String setting.

***class* saleae.analyzers.NumberSetting**(*, min_value=None, max_value=None, **kwargs)

Number setting, with an option min_value/max_value.

***class* saleae.analyzers.ChoicesSetting**(choices, **kwargs)

Choices setting.