Speck™ Development Kit Manual

Dec 2025

The information contained herein is for informational purposes only, and is subject to change without notice.

Intellectual Property Rights
SynSense owns the copyrights, trademarks and other intellectual property rights and interests in this document. The fact that SynSense provides this document to you does not affect the rights and interests of SynSense as described above.
Brand and product names are trademarks or registered trademarks of their respective owners.
No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document.

No Warranty
While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and SynSense is under no obligation to update or otherwise correct this information. SynSense makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of SynSense hardware, software or other products described herein.

Disclaimer
To the extent permitted by applicable law, SenSense shall not be liable for any direct, indirect, incidental, special, incidental or other damages, costs, liabilities or claims of any kind arising out of or in connection with the use of this document,with respect to the operation or use of SynSense hardware, software or other products described herein.

Applicable Terms and Conditions for Products
Terms and limitations applicable to the purchase or use of SynSense's products are as set forth in a signed agreement between you and SynSense or in SynSenseIs Standard Terms and Conditions.

Ⓒ SynSense — All rights reserved

Content

List of Figures

List of Tables

1. Introduction

1.1. Speck™ Chip

Speck^{TM} is a "sensor-compute integrated" neuromorphic intelligent dynamic vision System on Chip(SoC), integrating an asynchronous neuromorphic dynamic vision processor (DYNAP^{TM}CNN) and a Dynamic Vision Sensor (DVS), also known as an Event Camera. It features a large-scale Spiking Convolutional Neural Network (sCNN) chip architecture based on an asynchronous logic paradigm, configurable with up to 320K spiking neurons.

Speck™ is designed for always-on IoT devices and applications such as, human behavior recognition, gesture recognition, facial detection, and surveillance, with ultra-low power consumption and ultra-low latency.

1.2. Speck™ Development Kit

As is shown in **Figure 1**, Speck™ Development Kit(Dev Kit) is powered by the Speck™ chip, which provides a computational platform for user to interact with Speck™ via a host machine and software **Samna**. With the CQFP80 packaged Speck™ chip, user can mount lens based on expected vision field (the dev kit comes with 1.7mm and 3.6mm M12-Mount lens). With the vision module packaging, it comes with a built-in 1.98mm lens.

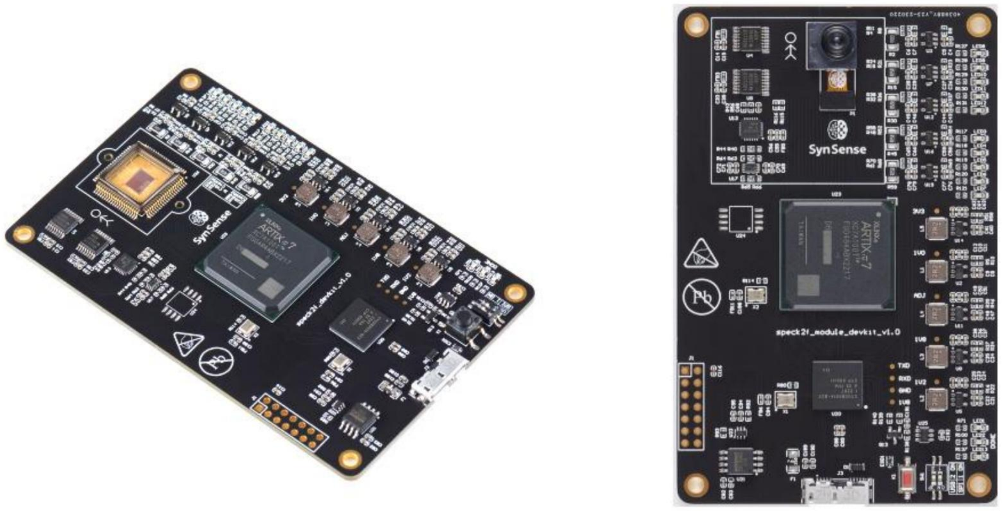


Figure 1 . Speck^{TM} Development Kit; **Left:** CQFP Packaging; **Right:** Camera Module Packaging

2. Speck™

2.1. Block Diagram

Figure 2 illustrates the top level block diagram of the Speck^{TM}. Between these blocks, only Address Event Representation(AER) communication protocol is used for data transmission. The internal DVS provides the sensory data inputs to event pre-processing and filter function block. The output interface, input interface and main computing resource(DYNAP^{TM}CNN cores) are sharing a same event router.

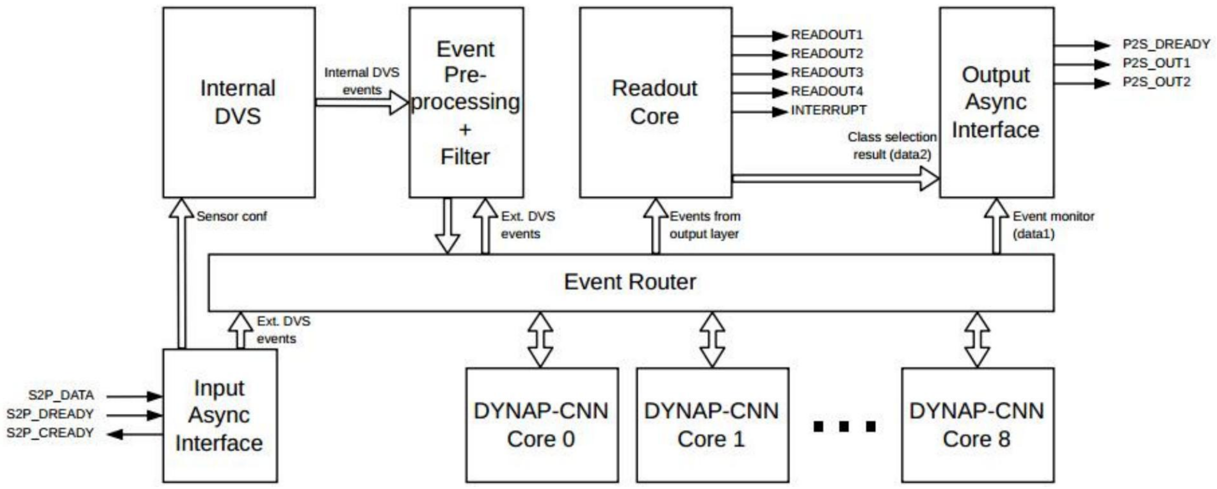


Figure 2． Top Level Chip Diagram

2.2. Key Features

. 1 Built-in DVS event pre-processing layer

. 9 DYNAP^{TM}CNN computational layers

. 1 Readout layer

. 128x128 DVS pixel array, dynamic range not less than 80 dB (20-200k lux)

Note: When DVS is used between 20lux to 50lux, there is a slight decrease pf the sensitivity of the optical sensor array.

2.3. Event Pre-processing Layer

. Noise filtering

. DVS polarity adjustment

. ROI selection

. Mirroring in both X/Y

. Rotate in 90-degree steps

2.4. DYNAP^{TM}CNN Computational Layers

. Up to 9 DYNAP^{TM}CNN layers

. Max input dimension 128*128

. Max feature output size 64*64

. Max feature number 1024

. Weight resolution 8 bits

. Neuron state resolution 16 bits

. Max kernel size 16*16

. Stride {1,2,4,8} independent in X/Y

. Padding [0..7] independent in X/Y

. Pooling 1: 1, 1:2, 1:4

. Fan-out of 2

. Linear Leak operation on each layer

. Spike count decimator on each layer

. Spike congestion balancer on each layer

. Parallel computing on layer 0 and layer 1, enabling larger throughput, can be used as input layers

2.5. Readout Layer

. 15 classes and 1 idle class

. Configurable moving average between [1, 16, 32] time steps

. 4 readout modes: inactive/threshold/max spiking class/specific class

. 4 readout pins and 1 interrupt pin

3. Getting Started Guide

3.1. Create a python environment

The Python-based tools for Speck development require a Python version between 3.8 and 3.12 inclusive. We recommend to use a python environment management utility such as conda or pyenv.

To create a compatible python environment using conda, use the following command in a terminal:

```
conda create --name speck 'python<=3.12'
```

3.2. Install requirements and dependencies

Development for Speck is performed using the open-source “sinabs” library from SynSense. To install the library and all required dependencies, use the following commands in a terminal:

```
conda activate speck
pip install sinabs
```

Set up UDEV rules on Linux

To connect with the Speck dev kit on Linux, you need to configure the UDEV rules on your system. Follow the instructions at the link below:
https://synsense-sys-int.gitlab.io/samna/0.48.0/install.html#udev-rules-on-linux-systems

3.3. Connect to the Speck dev kit

Important: The Speck dev kit requires a USB 3.1 connection to your PC. Make sure you use a USB3.1 cable (ideally the cable supplied with Speck), and avoid connecting to Speck through a USB hub. Also ensure that the USB port you use to connect provides the USB 3.1 standard.
Open Python in your python environment

```
] conda activate speck
] python

>>> import sinabs.backend.dynapcnn.io as sio
>>> sio.get_device_map()
{'speck2fdevkit:0': device::DeviceInfo(serial_number=, usb_bus_number=1, usb_device_address=1,
logic_version=0, device_type_name=Speck2fDevKit)}
```

If you don’t see output similar to the line above:

- If on linux, did you enable the UDEV rules?
- Are you using a USB3.1 cable and USB3.1 port? Check with lsusb (on Linux) or System Information on MacOS

3.4. Visualise the Speck sensor output

With these steps you can visualise the output of the Speck vision sensor in real time. Open python in your python environment, and run the following code block:

```python
import time

import sinabs.backend.dynapcnn.io as sio
devkit = sio.open_device("speck2fdevkit:0")

import samna
samna_graph = samna.graph.EventFilterGraph()
_, _, streamer = samna_graph.sequential(
    [
        devkit.get_model_source_node(),   # Specify the source of events to this graph as the devkit
        "Speck2fDvsToVizConverter",  # Convert the events to visualizer events
        "VizEventStreamer",   # Stream events to a visualizer via a streamer node
    ]
)
visualizer_port = "tcp://0.0.0.0:40001"

# Launch visualizer
gui_process = sio.launch_visualizer(
    receiver_endpoint=visualizer_port, disjoint_process=True
)
time.sleep(1.0)

# Visualizer configuration branch of the graph.
visualizer_config, _ = samna_graph.sequential(
    [samna.BasicSourceNode_ui_event(), streamer]  # For generating UI commands
)

# Connect to the visualizer
streamer.set_streamer_destination(visualizer_port)
if streamer.wait_for_receiver_count() == 0:
    raise Exception(f"Connecting to visualizer on {visualizer_port} fails.")
```

```
# Specify which plot is to be shown in the visualizer
plot1 = samna.ui.ActivityPlotConfiguration(
    image_width=128, image_height=128, title="DVS Layer", layout=[0, 0, 1, 1]
)
visualizer_config.write([samna.ui.VisualizerConfiguration(plots=[plot1])])
time.sleep(1.0)
samna_graph.start()
time.sleep(1.0)
devkit_config = samna.speck2f.configuration.SpeckConfiguration()

# enable monitoring the inputs from the DVS sensor
devkit_config.dvs_layer.raw_monitor_enable = True

# Apply this configuration
devkit.get_model().apply_configuration(devkit_config)
print("Press ENTER to terminate...")
input()

# Stop the graph
samna_graph.stop()

# If we used a sub-process to launch the visualizer, use that to terminate the visualizer.
if gui_process:
    gui_process.terminate()
    gui_process.join()
```

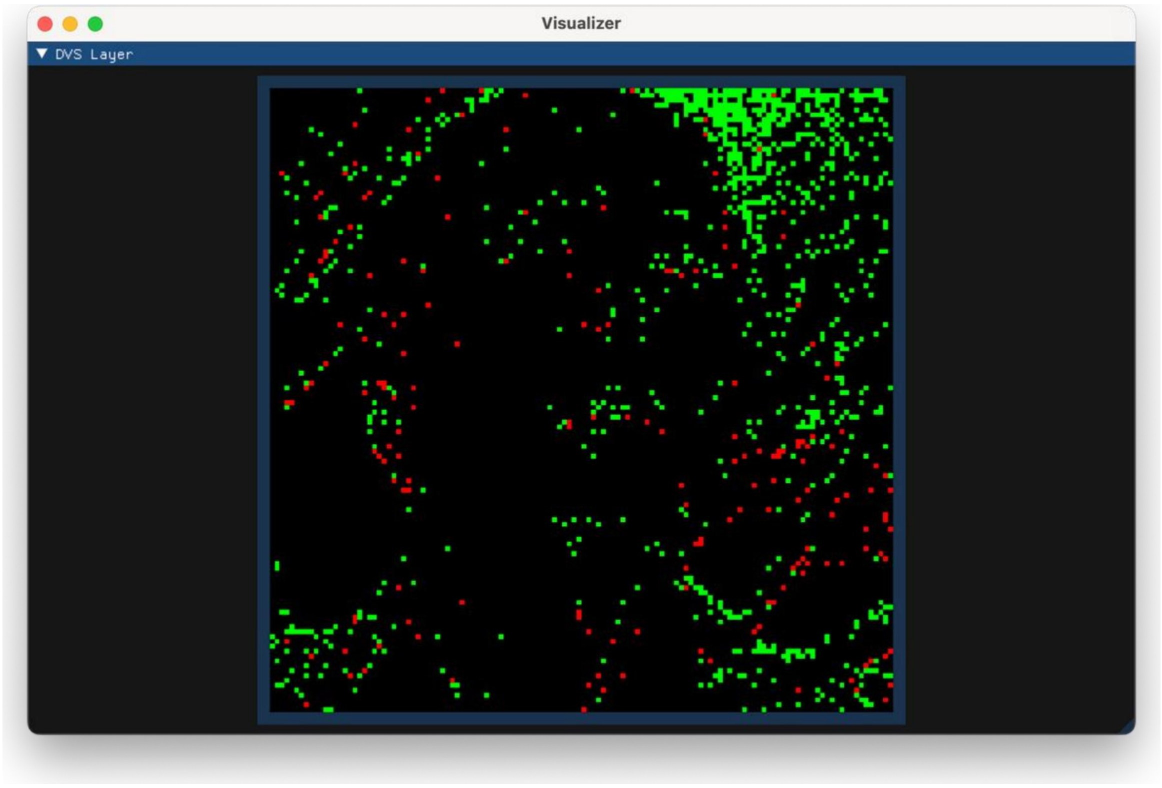


Figure 3 . The visualizer window, showing a live feed from the Speck sensor

3.5. Record and display some data from the Speck sensor, in Python

This code needs the plotting and visualization library matplotlib. To install this in your python environment, use the follow commands at the terminal:

```
conda activate speck
pip install matplotlib
```

Then open python, and paste the following code block:

```
import sinabs.backend.dynapcnn.io as sio
import samna
import time
import numpy as np
from typing import List
import matplotlib.pyplot as plt

# - Open the Speck devkit device
devkit = sio.open_device("speck2fdevkit:0")

# - Get a sink node to read events from the devkit
sink = samna.graph.sink_from(devkit.get_model_source_node())
```

```
# - Configure the Speck devkit to monitor DVS events
# Get a Speck 2f configuration, and enable input monitoring
devkit_config = samna.speck2f.configuration.SpeckConfiguration()
devkit_config.dvs_layer.raw_monitor_enable = True

# Apply this configuration to the Speck device
devkit.get_model().apply_configuration(devkit_config)

# - Enable timestamping on the dev kit
stopwatch = devkit.get_stop_watch()
stopwatch.start()
stopwatch.reset()

# - Define function to convert events into a numpy array
def dvs_events_to_numpy(events: List[samna.speck2f.event.DvsEvent]) -> np.ndarray:
    return np.array(
        [
            (ev.x, ev.y, ev.p, ev.timestamp)
            for ev in events
            if isinstance(ev, samna.speck2f.event.DvsEvent)
        ],
        dtype = [('x', 'u1'), ('y', 'u1'), ('p', bool), ('t', 'u4')],
    )
```

Now you can copy/paste and run the following code to accumulate and display events into frames of 1 second duration. You can repeat this code block as many times as you like.

```
# - Record events for 1 second
sink.clear_events()
time.sleep(1.)
events = dvs_events_to_numpy(sink.get_events())

# - Accumulate events into a frame
frame = np.zeros((128, 128))
np.add.at(frame, (events['x'], events['y']), 1.)

# - Show the frame
plt.imshow(frame.T)
plt.show()
```

3.6. Create and deploy a model to Speck

These steps show you how to deploy a simple model to Speck, to run on the dev kit.

```
import torch
import torch.nn as nn
from typing import List
from sinabs.from_torch import from_model
from sinabs.backend.dynapcnn import DynapcnnNetwork

ann = nn.Sequential(
  nn.Conv2d(1, 20, 5, 1, bias=False),
  nn.ReLU(),
  nn.AvgPool2d(2,2),
  nn.Conv2d(20, 32, 5, 1, bias=False),
  nn.ReLU(),
  nn.AvgPool2d(2,2),
  nn.Conv2d(32, 128, 3, 1, bias=False),
  nn.ReLU(),
  nn.AvgPool2d(2,2),
  nn.Flatten(),
  nn.Linear(128, 500, bias=False),
  nn.ReLU(),
  nn.Linear(500, 10, bias=False),
)

# Convert your model to SNN
sinabs_model = from_model(ann, batch_size=1, add_spiking_output=True)  # Your sinabs SNN model

# Convert your SNN to `DynapcnnNetwork`
hw_model = DynapcnnNetwork(
sinabs_model.spiking_model,
discretize=True,
input_shape=(1, 28, 28)
)

# Deploy model to a dev-kit
hw_model.to(device="speck2fdevkit:0")
```

The Python terminal should show this output:

```
Out[6]:
DynapcnnNetwork(
  (sequence): Sequential(
        (0): DynapcnnLayer(
          (conv_layer): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1), bias=False)
           (spk_layer): IAFSqueeze(spike_threshold=Parameter containing:
          tensor(636.), min_v_mem=Parameter containing:
          tensor(-636.), batch_size=1, num_timesteps=-1)
           (pool_layer): SumPool2d(norm_type=1, kernel_size=(2, 2), stride=None, ceil_mode=False)
        )
        (1): DynapcnnLayer(
           (conv_layer): Conv2d(20, 32, kernel_size=(5, 5), stride=(1, 1), bias=False)
           (spk_layer): IAFSqueeze(spike_threshold=Parameter containing:
          tensor(11364.), min_v_mem=Parameter containing:
          tensor(-11364.), batch_size=1, num_timesteps=-1)
           (pool_layer): SumPool2d(norm_type=1, kernel_size=(2, 2), stride=None, ceil_mode=False)
        )
        (2): DynapcnnLayer(
           (conv_layer): Conv2d(32, 128, kernel_size=(3, 3), stride=(1, 1), bias=False)
           (spk_layer): IAFSqueeze(spike_threshold=Parameter containing:
          tensor(8621.), min_v_mem=Parameter containing:
          tensor(-8621.), batch_size=1, num_timesteps=-1)
           (pool_layer): SumPool2d(norm_type=1, kernel_size=(2, 2), stride=None, ceil_mode=False)
        )
        (3): DynapcnnLayer(
           (conv_layer): Conv2d(128, 500, kernel_size=(1, 1), stride=(1, 1), bias=False)
           (spk_layer): IAFSqueeze(spike_threshold=Parameter containing:
          tensor(5748.), min_v_mem=Parameter containing:
          tensor(-5748.), batch_size=1, num_timesteps=-1)
        )
        (4): DynapcnnLayer(
           (conv_layer): Conv2d(500, 10, kernel_size=(1, 1), stride=(1, 1), bias=False)
           (spk_layer): IAFSqueeze(spike_threshold=Parameter containing:
          tensor(2841.), min_v_mem=Parameter containing:
          tensor(-2841.), batch_size=1, num_timesteps=-1)
        )
      )
    )
```

Now you can send some events to Speck to run through your deployed model:

```
# - Format data as required to input to Speck
def x_y_t_p_to_xytp(x: np.array, y: np.array, t: np.array, p: np.array) -> np.array:
    length = np.size(x)
    array = np.zeros(length, dtype = [('x', 'u1'), ('y', 'u1'), ('t', 'u4'), ('p', bool)])
    array['x'] = np.round(x)
    array[lyl] = np.round(y)
    array[ltl] = np.round(t)
    array[lpl] = p.astype(bool)

return array

# - Generate some random events
xytp = x_y_t_p_to_xytp(np.random.randint(0, 28, 10), np.random.randint(0, 28, 10), 0., np.array(True))


# - Get a chip factory to be able to convert events
cf = sinabs.backend.dynapcnn.chip_factory.ChipFactory(lspeck2fdevkitl)

# - Convert random event array to list of Speck events
input_evs = cf.xytp_to_events(xytp, 0, reset_timestamps=True)

# - Pass events to model running on Speck
hw_model(input_evs)
```

3.7. Next steps

- Train a spiking CNN for MNIST:
  https://sinabs.readthedocs.io/v3.1.0/tutorials/nmnist.html
- Deploy an MNIST model to Speck for inference:
  https://sinabs.readthedocs.io/v3.1.0/tutorials/nir_to_speck.html

3.8. Detailed tutorials

3.8.1 Enable Linear Leak Feature

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/leak_neuron.html

3.8.2 Event Pre-processing Layer

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/play_with_speck_dvs.html

3.8.3 Spike Count Visualization

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/visualize_spike_count.html

3.8.4 Power Monitor

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/power_monitoring.html

3.8.5 DYNAP™CNN Visualizer

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/visualizer.html

3.8.6 Using Readout Layer

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/using_readout_layer.html

3.8.7 Constraints - Available Network Architecture

https://synsense.gitlab.io/sinabs-dynapcnn/faqs/available_network_arch.html

3.8.8 Constraints - Available Operations

https://synsense.gitlab.io/sinabs-dynapcnn/faqs/available_algorithmic_operation.html

3.8.9 Chip Output Monitoring

https://synsense.gitlab.io/sinabs-dynapcnn/faqs/output_monitoring.html

4. Speck™ Development Kit

4.1. Mechanical Specification

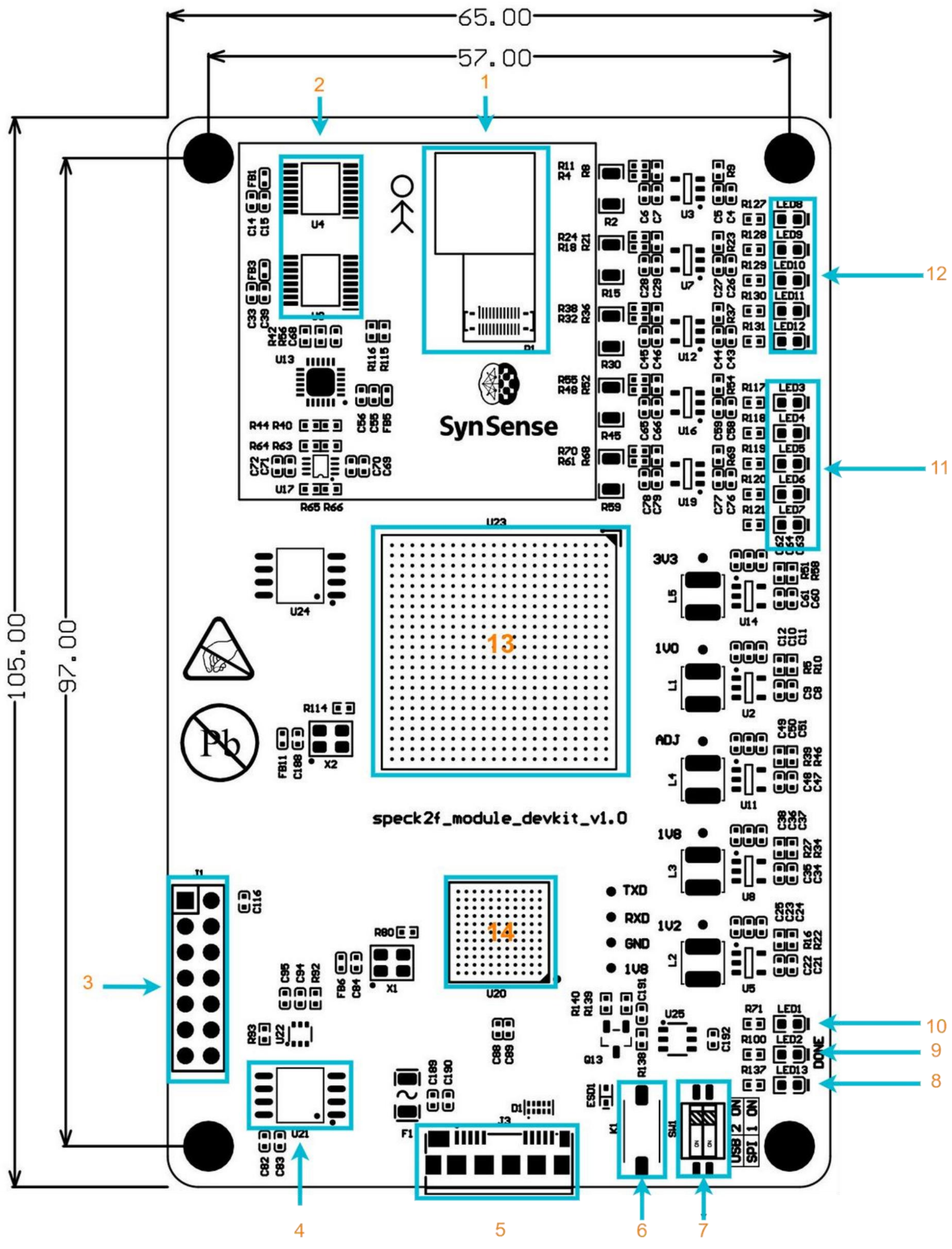


Figure 4 . Speck^{TM} Development Kit Front View

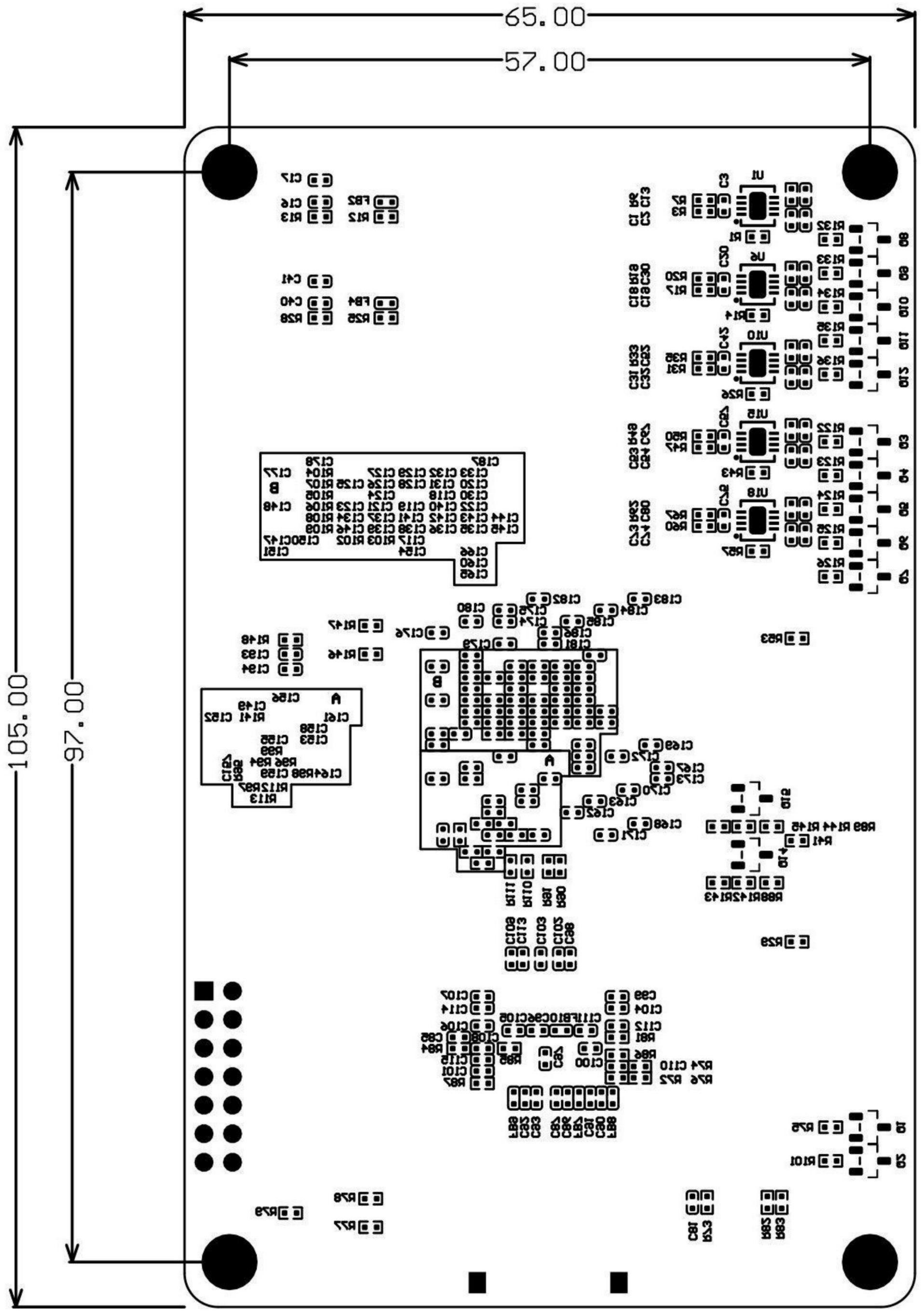


Figure 5 . Speck^{TM} Development Kit Back View

1. Speck™ SoC
2. High Precision Power Monitor
3. FPGA JTAG (RSV)
4. Flash
5. USB 3.0 Micro-B Port / Connector
6. System Reset Button
7. USB 3.0 Controller CFG Switch (RSV)
8. System Power LED
9. FPGA Configuration Done Indicator
10. USB 3.0 Controller State Indicator
11. Debug State Indicator
12. SoC Power Traces State Indicator
13. FPGA
14. USB 3.0 Controller

4.2. Development Data Sheet

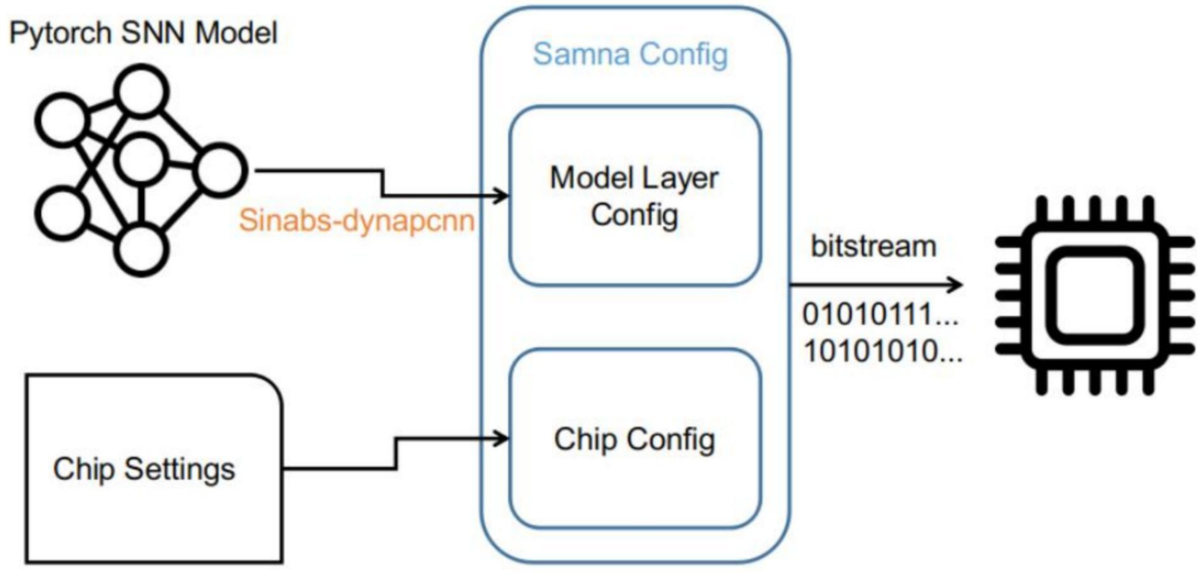


Figure 6 . Explanation of Samna Configuration

This chapter illustrates the detailed function block of Speck^{TM} and how these can be configured with the development kit and host machine software **samna** (see section **4.3** for more details).

The chip is typically configured using a bitstream that represents the configuration of individual registers. For high-level interaction with the chip, samna provides all essential APIs, eliminating the need for users to be concerned about low-level register addresses and bitstream generation.

As is shown in **Figure 5**. The samna configuration profile contains the model layer config(SCNN architecture and parameters) and the chip config.

For the model layer config, we provide an efficient software tool **sinabs-dynapcnn(**section **4.2)** that supports converting a sinabs/pytorch model to chip compatible model. This stage usually contains the translation of pre-trained network to chip resources mapping and the quantization of network parameters.

Regarding the chip configuration, it is typically intended for advanced users who wish to implement their customized chip settings. This involves manually modifying neuron dynamics, connectivity between DYNAP^{TM}CNN layers, event-preprocessing layers, and readout layers.

4.2.1 Preliminary Software

The Speck^{TM} Dev Kits build essential resources to support the user interact with Speck^{TM}.

Begin by following the Getting Started guide in section 3.

4.2.2 Samna Configuration

With a pre-trained neural network, the hardware configuration for Speck can be generated via **sinabs-dynapcnn** software with:

```
dynapcnn_network = DynapcnnNetwork(snn=snn, discretize=True, dvs_input=True, input_shape=(1, 128, 12
8))
samna_cfg = dynapcnn_network.make_config(device="speck2fmodule")
# Note: The device name str can be different depends on the devkit. The example shows the speck2f dev-kit
with module packaging
```

Once this configuration object is generated, it contains the network architecture profile and parameters from the snn. The configuration also contains all the chip settings that allows user to manually define their implementation. The main structure of this configuration file is shown as follow:

```
samna_config
    -cnn_layers  # stands for DYNAPCNN layers
        -0
            -biases
            -destinations
            -dimensions
            -input_congestion_banlancer_enable
            -leak_enable
            -leak_internal_slock_clk_enable
            -monitor_enable
            -neurons_initial_value
            -output_decimator_enable
            -output_decimator_interval
            -return_to_zero
            -threshold_high
            -threshold_low
            -weights
        -1
        ...up to 8
    -dvs_filter  # exsist in event-preprocessing layer
        -enable
        -filter_size
        -hot_pixel_filter_enable
        -internal_slow_clk
        -low_pass_mode_enable
        -threshold
    -dvs_layer  # exsist in event-preprocessing layer
        -cut
        -destination
        -merge
        -mirror
        -mirror_diagonal
        -monitor_enable
        -off_channel
        -on_channel
        -origin
        -pass_sensor_events
        -pooling
        -raw_monitor_enable
    -readout
        -enable
        -internal_slow_clk
        -low_pass_filter32_not16
        -low_pass_filter_disable
        -monitor_enable
        -output_mode_sel
        -output_neuron_num
        -override_threshold_max
        -readout_configuration_sel
        -readout_pin_monitor_enable
        -threshold
```

4.2.3 Event Pre-processing Layer

The general event pre-processing pipeline is shown as **Figure 6**, the function block on the board is executed exact in this sequence order.

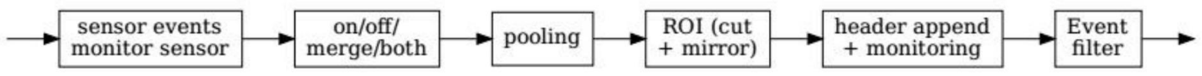


Figure 7 . Computation Pipeline of Event Pre-processing Layer

For the hands-on tutorial of how to use the layer, please refer to section **5.5.**

4.2.3.1. On/Off/Both/Merge Switching

The events generated by DVS is featured with two polarity on/off, which indicates the actual light intensity change from low-high and high-low respectively. The two channel can be configured flexibly as:

Merge two polarities as one channel(Sum):

```
samna_config.dvs_layer.merge = True
```

Keep only one channel:

```
# keep only on channel
samna_config.dvs_layer.off_channel = False
samna_config.dvs_layer.on_channel = True
```

```
# keep only off channel
samna_config.dvs_layer.off_channel = True
samna_config.dvs_layer.on_channel = False
```

4.2.3.2. Pooling

The pooling in event pre-processing layer is simply implemented by mapping mechanism between source neurons and target neuron. This is done by integrating the events from Region of Interest(ROI) to destination. In pre-processing layer, pooling supports the kernel size of [1, 2, 4] for both X and Y axis, the stride is default to set the **same** with the kernel size.

Example of configure a 2x2 stride of 2 pooling on pre-processing layer:

```
samna_config.dvs_layer.pooling.x = 2
samna_config.dvs_layer.pooling.y = 2
```

4.2.3.3. ROI selection

The ROI selection is designed for user to freely use the resolution of internal DVS. The function can be used to select a rectangle region with defined top-left and bottom-right corner coordinate. The result ROI will be automatically shift to top-left start from (0, 0). In software, it is set the object *origin* stands for top-left corner and *cut* stands for bottom-right corner. An example as shown in **Figure 7** for select the center input of 64x64 region is:

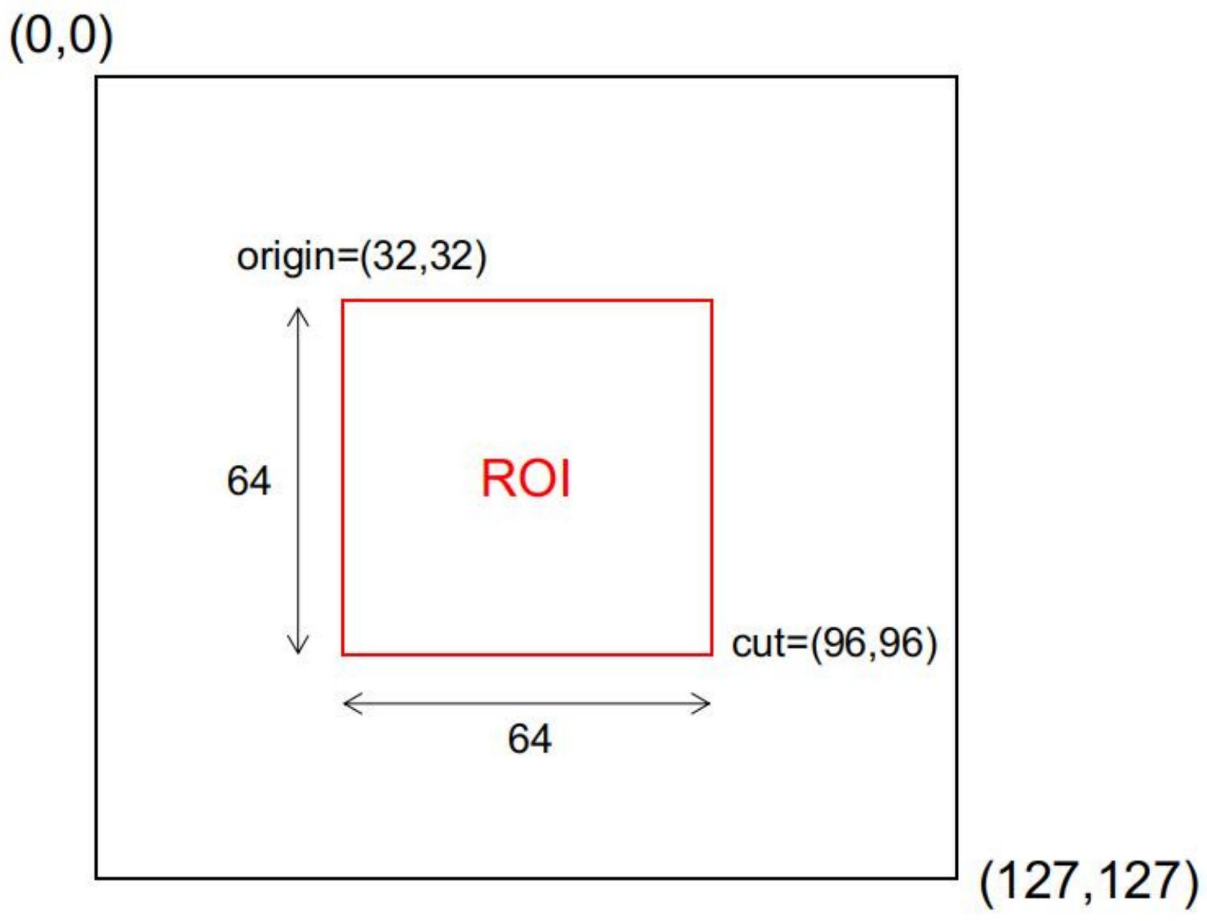


Figure 8 . Indication of ROI Selection Example

```
samna_config.dvs_layer.origin.x = 32
samna_config.dvs_layer.origin.y = 32
samna_config.dvs_layer.cut.x = 96
samna_config.dvs_layer.cut.y = 96
```

4.2.3.4. Mirror Operation

The mirror enables the user apply flip based on pixel coordinates. This including flip horizontally(along y axis), vertically(along x axis) as well as swapping the axis between x and y.

By using samna:

```
# Horizontal Flipping
samna_config.dvs_layer.mirror.x = True

# Vertical Flipping
samna_config.dvs_layer.mirror.y = True

# Swap X and Y axis
samna_config.dvs_layer.mirror_diagonal = True
```

4.2.3.5. DVS Event Filter

The built-in filter supports 3 types of the filtering operation allows the user to flexibly configure the denoising requirements based on DVS input.

These are:

- DVS Filter(shot noise filter)
- Low Pass(flicker noise filter)
- Hot Pixel Filter + DVS filter

4.2.3.5.1. DVS Filter(Noise Filter)

The DVS filter block included in the pre-processing layer in order to filter the neighboring sparse noisy activity. In general, an event at a position (x,y) is forwarded by the filter when at least one pixel in the vicinity of (x,y) has spiked in a defined time window before this event.

Whenever a pixel event arrives at the filter, the filter stores its timestamp and coordinate in a memory space. Then when a new pixel event arrives, the filter checks the pre-defined area in a number of clock-cycles. The actual timestamp are recorded using a counter and each count is triggered by the slow-clock(**3.2.6**) source(for more information please check the slow-clock section).

The filter can be configured in follow aspects:
To enable the filter:

```
samna_config.dvs_filter.enable = True
```

a. Filter Window Size

The window size defines the area of the neighboring in the spatial domain. The filter size can go from 1x1 to 15x15.

```
# setting a 3x3 filter that checking its surroundings
samna_config.dvs_filter.filter_size.x = 3
samna_config.dvs_filter.filter_size.y = 3
```

b. Filter Delta/Threshold

The Delta threshold of filter is used to compare the current counter value with the value of pixels neighboring the current activated pixel. If any of the neighboring pixels has a value difference less than the Delta value, the filter will let current spike pass through, other wise the current event is blocked.

```
# setting the Delta threshold to 2
samna_config.dvs_filter.enable = True
samna_config.dvs_filter.threshold = 2
```

4.2.3.5.2. Low Pass(Flicker Filter)

In this mode, the filter works as a low-pass filter. This mode can be potentially used to filter out 50/60Hz light flicking noise produced by some lights such as fluorescent or LEDs.

Low-pass filter is used to filter the events which the time interval below a certain threshold, solving the flickering problems. If there is another event in a certain period (which depends on slow clock rate and threshold), the event would be regarded as noise and be filtered. The usage of this filter is also highly depends on the setting of slow-clk. e.g. slow-clock rate is 1000Hz(1ms) and threshold is 25, so the period is 1x25 =25 ms, which means if the time interval of two events in same pixel is lower than 25 ms, the event would be filtered. So in flickering environment, those high-frequency flickering noise would be filter after enabling low-pass filter mode.

To enable the filter:

```
samna_config.dvs_filter.enable = True
samna_config.dvs_filter.hot_pixel_filter_enable = False
samna_config.dvs_filter.low_pass_mode_enable = True
```

Then the filter size should be set to 0 and set the threshold

```
samna_config.dvs_filter.filter_size.x=0
samna_config.dvs_filter.filter_size.y=0
samna_config.dvs_filter.threshold = 25
```

4.2.3.5.3. DVS filter + Hot Pixel Filter

The hot pixel filter is used to filter the expected high frequency signal that typically caused by the manufacture mismatch of DVS circuitry. The hot pixel usually is fixed to a location and has a firing rate of 50-1000Hz. The threshold is also highly depends on the setting of external slow-clk(section **3.2.6**).

To use the Hot Pixel Filter

```
samna_config.dvs_filter.enable = True
samna.config.dvs_filter.hot_pixel_filter_enable = True
samna.config.dvs_filter.threshold = 5 # setting up the threshold
```

4.2.3.5.4. Explanation of filter mode conflicts and slow-clock

Due the restriction of hardware resources, The filter mentioned in 3.2.4.5.1-3 can not be fully implemented at the same time. The user should only configure the filter operation in one of the mode:

- Low Pass Mode
- DVS Filter model
- DVS Filter + Hot Pixel Filter Mode

Since the filter performance is highly depended on the timing reference, the setting of the external/internal slow-clock is crucial for the correct usage of these filtering techniques. The development kit with samna provides an efficient way for user to set up the slow clock, for further information, please check section **3.2.6**.

4.2.3.6. Fan-out

The event pre-processing layer can maximally have fan out of two. This stands that the output event from the layer can be copied and forward to 2 different destination. By default, the fan-out of pre-processing layer is set to be fed into the first layer of designed neural network.

To set multiple destination of the layer:

```
# setting the output to DYNAPCNN core 4
samna_config.dvs_layer.destinations[0] = 4
```

```
# setting a copy of output to DYNAPCNN core 5
samna_config.dvs_layer.destinations[0] = 5
```

4.2.3.7. Monitoring

The monitor in event pre-processing layer enables the user to receive all the events that arrives here. When enabled, output messages of the DVS pre-processing block are forwarded on the monitor bus to the output serial interface. This is typically opened when user tends to output DVS events. To use it, simply turns the monitor on by:

```
samna_config.dvs_layer.monitor_enable = True
```

4.2.3.8. Disable the Event Pre-processing Layer

If disabled, the internal sensor events are dropped, i.e., not forwarded to the DVS pre-processing block. This is typically used when sensor data is provided by an external source (Ext DVS Mode) or feeding customized data from host machine.

```
# Disable the event from internal dvs and event-preprocessing layer
samna_config.dvs_layer.pass_sensor_event = False
```

4.2.4 DYNAPCNN Layers

The main computational resources of Speck™ are 9 configurable SCNN layers called DYNAPCNN layers or cores. As is shown in **Figure 8**, each of these layers can implement a sequence computation that **equivalent to [convolution->spiking neuron->pooling] structure**. **(Note all the term "layer" in the doc refer to a single DYNAPCNN layer/core).** Individual core can be connected to form a user defined network of any size up to the maximum available resources. Layer memory sizes are balanced to provide a flexible balance of resources, with larger or smaller layers. The data flow between core and core are purely based on Address Event Representation(AER) protocol, where only event signals are used in communication between layers.

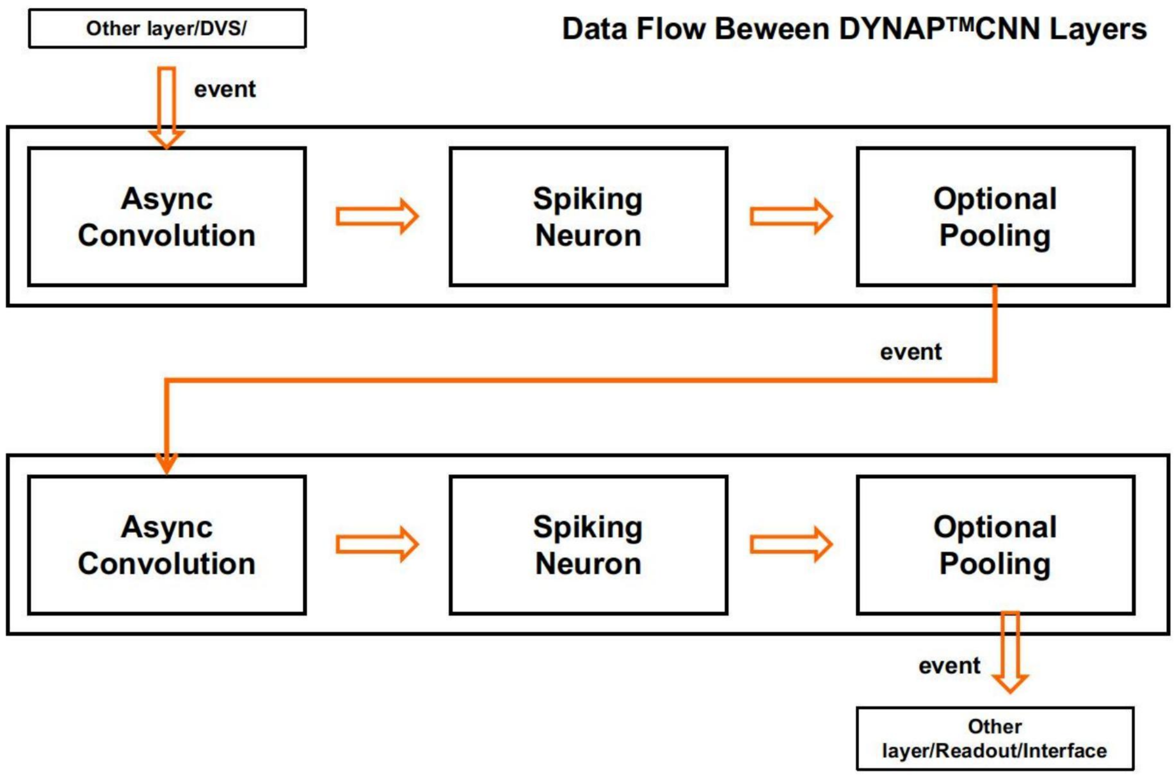


Figure 9． DYNAP™CNN layer Data Flow Pipeline

4.2.4.1. Memory Capacity and Resolution of DYNAPCNN Layers

The Speck™ is divided into 9 cores, each of which executes a single DYNAPCNN™ layer(core). The memory capacities of the cores are different, and restrict the implementation of larger layers to specific cores.

Table 1: DYNAP™CNN Memory Distribution

Core	Kernel memory (WORD)	Leak memory (WORD)	Neuron memory (WORD)
0	16 Ki	1 Ki	64 Ki
1	16 Ki	1 Ki	64 Ki
2	16 Ki	1 Ki	64 Ki
3	32 Ki	1 Ki	32 Ki
4	32 Ki	1 Ki	32 Ki
5	64 Ki	1 Ki	16 Ki
6	64 Ki	1 Ki	16 Ki
7	16 Ki	1 Ki	16 Ki
8	16 Ki	1 Ki	16 Ki

Table 2: DVS Event Filter Block Memory Capacity

SRAM	Filter memory (WORD)
DVS Event Filter	16 Ki

Table 3: Available Parameter Resolution

	Memory Type	Word Length
1	Kernel	8 bits
2	Neuron	16 bits
3	Leak	16 bits
4	Filter	16 bits

Let a network be defined by the number of input features c, the number of output features f, and the kernel dimensions k_x and k_y. The theoretical number of WORDs required for kernel memory K_M is then

$$K_M = c f k_x k_y$$

The total number of memory WORDs required is

$$K_{MT} = c \cdot 2^{\lceil \log_2 (k_x k_y) \rceil + \lceil \log_2 (f) \rceil}$$

The required number of neuron memory WORDs NM depends on the dimensions of the input features c_x and c_y, as well as the stride and padding s_x , s_y, and p_x, p_y.

$$f_x = \frac{c_x - k_x + 2p_x}{s_x} + 1$$

$$f_y = \frac{c_y - k_y + 2p_y}{s_y} + 1$$

$$N_M = f f_x f_y$$

Again the total number of required WORDs on the chip side is larger.

$$N_{MT} = f \cdot 2^{\lceil \log_2 (f_y) \rceil + \lceil \log_2 (f_x) \rceil}$$

Taking an example of convolutional layer

```
conv_layer = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3,3), stride=(1,1), padding=(1,1))
```

Assuming the input dimension of 64x64, the output feature map size can be obtained as:

$$f_x = \frac{64 - 3 + 2 * 1}{1} + 1 = 64$$

$$f_y = \frac{64 - 3 + 2 * 1}{1} + 1 = 64$$

The actual kernel memory entries is calculated thus:

$$k_{MT} = 16 * 32 * 4 * 4 = 8\mathrm{ki}$$

The actual neuron memory entries is then:

$$N_{MT} = 64 * 64 * 32 = 128\mathrm{ki}$$

Where 128Ki neuron exceeds any available neuron memory constrains among 9 layers, thus this layer ****CANNOT**** be deployed on the chip.

In addition to the neuron memory and kernel memory constraints, the hardware design limits few of dimensions in terms of convolutional layer settings as listed out in section **2.2**:

- For output channel number/feature number of the convolutional layer, maximally can be set to 1024.

```
# up to 1024
conv_layer = nn.Conv2d(in_channels=16, out_channels=1024, kernel_size=(3,3), stride=(1,1), padding=(1,1))
```

- For convolutional kernel size, maximally can be set to 16x16

```
# up to 16x16
conv_layer = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(16,16), stride=(1,1), padding=(1,1))
```

- For convolutional kernel stride, the available choice are {1,2,4,8}

```
# up to 8x8
conv_layer = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(16,16), stride=(1,1), padding=(1,1))
```

- For padding size, available choice are {0,1,2...7}

```
# up to 7x7
conv_layer = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(16,16), stride=(1,1), padding=(1,1))
```

- For pooling kernel size, available choice are {1x1, 2x2, 4x4}

```
# up to 4x4
conv_layer = nn.AvgPool2d(kernel_size=(4,4))
```

- For output feature map size after convolution, maximally accepts 64x64.

4.2.4.2. Congestion Balancer in DYNAPCNN Layers

In Speck™, DYNAPCNN layer has a congestion balancer block at its data path input. This is designed for reducing event bandwidth as user needed.

The congestion balancer enables dropping of input spikes at any time when the convolutional core of the layer is busy processing previous event. Specifically, if a train of spikes are sent to the layer, a number of them will be accepted (via some buffering) and the convolution computation starts. If, for example, the kernel is very large and a new spike arrives while the layer input is busy, this new spike will be dropped. As soon as the layer is again available, a coming spike will be processed.

This block is then able to adapt the spike input frequency to the convolution by capping it to the maximum that the layer can process. When disabled, the block will let all spikes through. This feature is controlled by the input_congestion_balancer_enable.

Example:

```
# Open congestion balancer for DYNAPCNN layer 0 and layer 1
samna_config.cnn_layers[0].input_congestion_balancer_enable = True
samna_config.cnn_layers[1].input_congestion_balancer_enable = True
```

4.2.4.3. Spike Decimator

For each DYNAPCNN layer, it is equipped with a decimator block at its **data path output**. The decimator block enables the user to reduce the spike rate at the output of a convolutional layer. When disabled, the block will let all spikes through as normal.

This feature is controlled by the output_decimator_enable with configurable choice from 2 to 512.

Table 4: Spike Decimator Settings

Decimator_interval	Description
0	1 spike passed every 2
1	1 spike passed every 4
2	1 spike passed every 8
3	1 spike passed every 16
4	1 spike passed every 32
5	1 spike passed every 128
6	1 spike passed every 256
7	1 spike passed every 512

Example:

```
# enable the decimator for DYNAPCNN core 3 and set 50% drop of its output events
samna_config.cnn_layers[3].output_decimator_enable = True
samna_config.cnn_layers[3].output_decimator_interval = 0
```

4.2.4.4. Convolution and Spiking Neuron Operation

In each DYNAP™CNN layer, whenever a spike arrives, it follows a async convolution, spiking neuron activation and pooling operation. As is shown in Figure 9, Speck™ is highly optimized with event-driven processing, the event-driven convolution does not operate on a frame basis but only happens when an event arrives at the convolution pipeline. When a spike with address information reaches an SNN core, the kernel value and destination neuron position are obtained by searching the address. Then, the neuron states are updated asynchronously based on the synaptic operation. Asynchronous convolution is not affected by the arrival of other input events or cores, so it can be efficiently distributed in parallel for multiple events at different spatial positions. If pooling is applied to the pipeline, it happens after the spiking neuron activation and implement the same as described in 3.2.3.2 which maps the destination to a same neuron.

For each channel in a layer, it shares a single 16bit int value for biases/leak(check section 3.2.4.7 for further detail). For each layer, it shares a single 16bit int value for threshold_high and threshold low for all neurons. For more restrictions of convolutional layer and spiking layer setting, check section 2.4 and section 3.2.4.1.

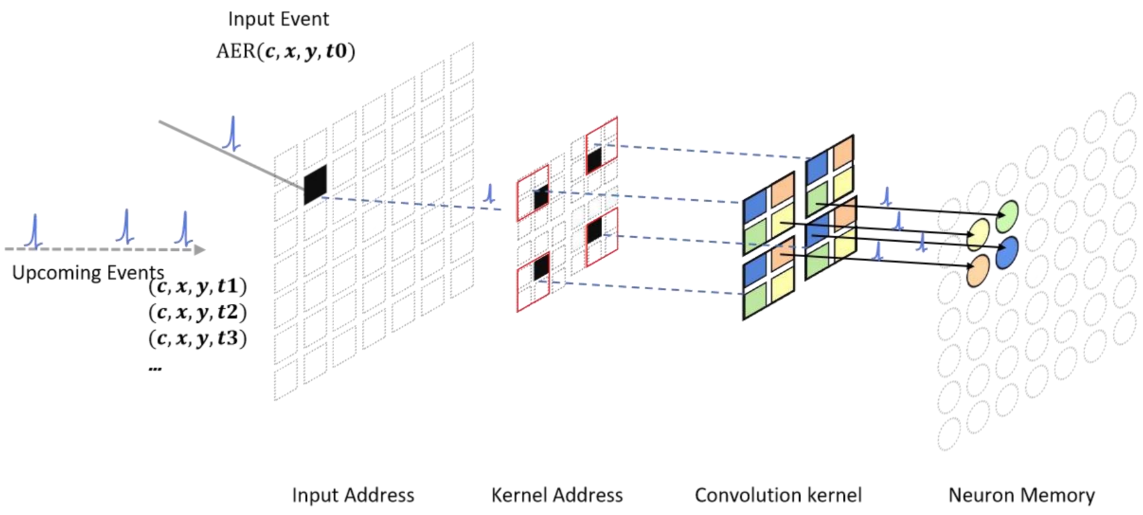


Figure 1 0 . Explanation of Asynchronous Convolution and Spiking Activation Operations

4.2.4.5. Network Embedding

Typically, with a pre-defined sequential SCNN network structure, it is suggested to implement the conversion via the built-in API from **sinabs-dynapcnn** package. The connectivity, parameter quantization and bitstream configuration can be automatically translated to samna configuration.

An example of converting the network from pytorch model:

```
network = nn.sequential([
                    nn.conv2d(),
                    IAFsqueeze(),
                    nn.pool(),
                    # up to here is using 1 dynapcnn layer
                    nn.conv2d(),
                    IAFsqueeze(),
                    nn.Flatten(),
                    nn.Linear(),
                    IAFsqueeze(),
                    # up to here is using 2 dynapcnn layer
                    ])
dynapcnn_network = DynapcnnNetwork(snn=network, discretize=True, dvs_input=True, input_shape=(1, 128,
 128))
samna_cfg = dynapcnn_netowrk.make_config(device="speck2fmodule")
```

However, user can still freely modify the network architecture by representation in the samna configuration:

```
# For a instance, checking the parameter of core 0
samna_config.cnn_layers[0].biases                  # bias parameter
samna_config.cnn_layers[0].weights                 # weight parameter
samna_config.cnn_layers[0].destinations          # connectivities
```

4.2.4.6. Neuron Dynamics

For each DYNAP™CNN layer, neuron can be set in several aspects to achieve different neuron dynamics

4.2.4.6.1. Neuron Initial State

By the default, the neuron initial membrane potential can be automatically translated into the configuration if the network configuration is converted from **sinabs-dynapcnn**. However, users are still able to manually define any neurons' initial membrane potential by setting values to object:

```
# For a instance, checking the parameter of core 0
samna_config.cnn_layers[0].neurons_initial_value                    # bias parameter
```

4.2.4.6.2. Membrane Reset Mechanism

For neuron reset mechanism, it refers to the operation for the neuron state/membrane potential when it emit a spike. Speck™ supports two ways of resting: hard reset and hot reset. This is controlled by return to zero.

The hard reset will reset the neuron membrane potential to 0 when there is a spike fires out:

```
# For a instance, setting the neurons of core 0
samna_config.cnn_layers[0].return_to_zero = True
```

The soft reset will instead subtract the membrane potential by 1 x pre-defined neuron firing threshold.

```
# For a instance, setting the neurons of core 0
samna_config.cnn_layers[0].return_to_zero = False
```

Note: Due to the asynchronous feature of neuron operation, the spiking neuron calculation is only triggered by once when a incoming spike arrives. This means if the membrane potential exceeds 2 x pre-defined threshold, the neuron will still fire only once. When soft reset applied, the neuron will still have more than 1 x threshold membrane potential left and neuron will remain silent until next input spike arrives.

4.2.4.6.3. Upper/Lower Firing Threshold

For hardware implementation of neurons, it is essential to limit the neuron membrane potential upper/lower limits since the memory cannot go infinity. The upper threshold acts the same role as the firing threshold where neuron will emit a spike when it exceeds the threshold. The lower threshold constraints the minimum membrane potential during the computation. When a negative activation applied, if membrane potential already stands at threshold_low, the membrane potential will not change. These are controlled by threshold_high and threshold_low respectively. These two parameter can also be defined in Sinabs.

To directly modify thresholds through samna configuration:

```
# For a instance, setting threshold for core 0
samna_config.cnn_layers[0].threshold_low = -100
samna_config.cnn_layers[0].threshold_high = 100
```

Important Notice: There is a known BUG in current generation of Speck^{TM} that threshold_low cannot be set equal to 0, Please alternatively choose a nearest value instead e.g. *threshold_low = -1.*

4.2.4.7. Leak/Bias Operation

The leak operation of neuron is designed independent of the asynchronous neuron calculations and it is only driven by a reference clock signal(slow-clock). For each DYNAP^{TM}CNN layer, it includes a leak generation block which will update all neuron values in a layer with provided leak values.

If use the bias operation, a calculated slow clk and bias value should be provided where in the **Samna Configuration**:

```
# For a instance, setting leak/bias for core 0
samna_config.cnn_layers[0].leak_enable = True

# using the internal DVS count based clock
samna_config.cnn_layers[0].leak_internal_slow_clk_enable = True

# using external clock
samna_config.cnn_layers[0].leak_internal_slow_clk_enable = False

# Set biases
samna_config.cnn_layers[0].biases = [-127] # assuming only one channel
```

4.2.4.8. Fan-out

For each DYNAP^{TM}CNN layer, same as described for event pre-processing layer in section **3.2.3.6**, it supports up to maximally 2 output destinations. This can be controlled by destinations.

```
# For instance setting output destination from core 0 to core 1 and core 2
samna_config.cnn_layers[0].destinations[0] = 1
samna_config.cnn_layers[0].destinations[1] = 2
```

This potentially increase the flexibility of network structure where user can configure different architecture as is shown in **Figure 10**.

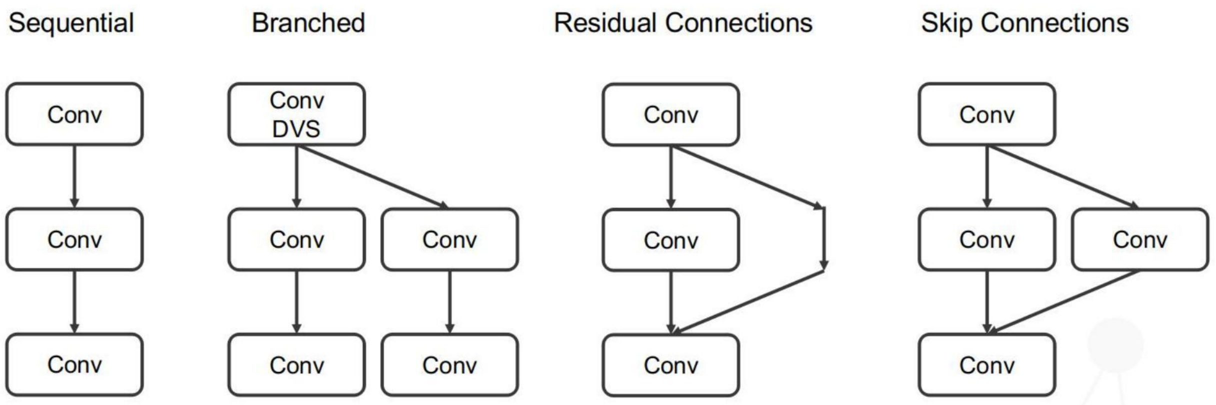


Figure 1 1 . Diagram of Available Network Structure

4.2.4.9. Monitoring

Same as described in **3.2.3.7**, the output event signal from each DYNAP^{TM}CNN layer can be monitored via the asynchronous interface together with host machine software **samna**. To enable the monitor of a single DYNAP^{TM}CNN layer:

```
# For instance setting monitor enable for core 0
samna_config.cnn_layers[0].monitor_enable = True
```

4.2.5 Readout Layer (Post Processing)

Hint: If user do not intend to build a hardware system that directly make use the interrupt signal with Speck™, it is not suggested to use the readout layer for on-chip post-processing. For a hands on tutorial for readout layer usage please check section **5.8.**

The main use of the post-processing block is to calculate the moving average over a time window for a maximum of 15 neurons, provide the maximum average of the 15 neurons and compare the value of the calculated moving averages against a specified threshold. 5 pins of Speck™ are dedicated to the direct readout of the class of maximum activity, these pins (INTERRUPT and READOUT1 to 4) (check block diagram in section 2.1) are designed to provide a direct readout of the maximum spiking class (with or without activity threshold). The readout pins are typically used when communication with an external computation platform such as MCU/FPGA etc.

4.2.5.1. INTERRUPT Pins

This pin outputs 0 until the class of max activity exceeds the *threshold*. Alternatively, the threshold comparison can be overridden by setting *override_threshold_max* to True. In this case, INTERRUPT becomes 1 at every falling edge of the slow-clk.

The INTERRUPT pin is raised at the falling edge of the slow-clk only if *override_threshold_max* is True or the max class activity is again above the selected threshold.

4.2.5.2. Readout Pins

There are 4 readout pins. READOUTx pins reflect the index of the class of max activity as described in Data Output Modes. These pins are activated in two cases:

. A class has spiked more than the set threshold during the previous readout clock period (INTERRUPT is also raised when this condition is met).

. The *override_threshold_max* is set to True (override threshold).

The 4 bits reflect the binary value of the most recent spiking class. As such, in an application requiring only 4 classes, the CNN can be configured such that the four output classes are encoded as class 1, 2, 4 and 8 when arriving at the readout layer. In this condition, the 4 output pins READOUTx will each directly reflect one of the classes of interest, and no decoder will be needed to interpret the chip output.

4.2.5.3. Readout Pin Monitoring

The readout layer in Speck^{TM} is the post-processing layer, the output results are readable through the 4 readout pins if an interrupt happens if configured correctly.

The readout pin monitoring feature can be enabled via **samna**. To enable the readout layer, the samna.speck2f.configuration.ReadoutConfig.enable needs to be set to True first. To forward your model's last layer to the readout layer, you need to set its destination to 12.

The samna.speck2f.configuration.ReadoutConfig.readout_configuration_sel needs to be set according to your model. There are 4 different addressing modes that could be selected:

Table 5: Readout Pin Mode Settings

Value	Mode
0	2x*2y*4f
1	2x*4y*2f
2	4x*4y*1f
3	1x*1y*16f

And set the samna.speck2f.configuration.ReadoutConfig.threshold of the readout layer according to your model. The moving average of the output neurons is compared to the threshold value to produce an output if the received number of spikes is greater than the threshold.

The Speck^{TM} readout layer also provides a low pass filter. There are two selectable time windows, 16 (16 * slow clk period) and 32 (32 * slow clock period), which can be chosen by samna.speck2f.configuration.ReadoutConfig.low_pass_filter32_not16.

The default value is False, which is 16 * slow clock period. The low pass filter is **enabled by default**, if you don't want to use it, please set:

samna.speck2f.configuration.Readout-Config.low_pass_filter_disable to True.

Then we set samna.speck2f.configuration.ReadoutConfig.readout_pin_monitor_enable to True in order to monitor the 4 readout pins.

If there is a valid result, an interrupt is generated by the chip and a samna.speck2f.event.ReadoutPinValue event is sent to Samna.

The samna.speck2f.event.ReadoutPinValue contains 2 members, an index, indicating the feature, and a timestamp in microsecond, indicating when this event happened.

4.2.5.4. Readout Time Window

The time window where the moving average is calculated is configurable according to the clock provided by external slow-clock, and can have time window of 1, 16 and 32 times the provided clock rate. The output data from the readout block can be extracted by using different configuration modes. Moreover, some timing characteristics of the block and the addressing mode of the neurons are configured.

4.2.5.5. Data Output Modes

The *ReadoutValue* is generated at every slow-clock cycle, it has an attribute named "value" which is a 21 bits data, as is shown in **Table 6**, it could have different meaning

Table 6: Readout Mode Selection Settings

output_mode_sel	bit[20]	bit[19:16]	bit[15:0]
0	data valid	neuron index of max	power down (clock gating)
1	data valid	neuron index of max	threshold compare output
2	data valid	neuron index of max	average output of the selected neuron
3	data valid	neuron index of max	average output of max spiking neuron

- if *output_mode_sel* is set to 0, the data_out is equal to 0.
- if *output_mode_sel* is set to 1, data_out[15:0] consists of the data of the threshold comparison, the index of the maximum moving average neuron and the data valid signal. The threshold comparison data is the 16 bit value of the comparison of each neurons moving average with the threshold.
- if *output_mode_sel* is set to 2, data_out[15:0] consists of the moving average of the selected neuron, the index of the maximum moving average and the data valid signal. The data valid signal is asserted after all the computations have finished in order to ensure correct sampling of the data.
- if *output_mode_sel* is set to 3, data_out[15:0] consists of the maximum average of the 16 neurons, the index of the maximum moving average neuron and the comparison output between the maximum moving average and the threshold.

4.2.6 Slow Clock

The slow-clock is internally used/externally provided to Speck^{TM} to operate a number of features.

4.2.6.1. Clock Speed

In a typical application, the slow-clock toggles at a speed of around 10Hz to 10kHz, the frequency depends on the internal use of the clock and on the specific application.

4.2.6.2. Usage

The slow-clock unifies three timing sources used by different functional blocks

- Leak/Bias Clock (section **3.2.4.7**): Each DYNAP^{TM}CNN layer including a leak circuitry receive the slow-clk to trigger a leak operation at every clock cycle.
- DVS Filter Block (section **3.2.3.5**): The DVS event filters use the slow-clock to provide the timing reference and update the internal states.
- Readout Block (section **3.2.5**): The readout layer uses the slow-clock as the timing reference for moving-average clock to time the calculation of output class moving averages.

Important Notice: The three function block are sharing the same slow-clock source.

4.2.6.3. Generation

The slow-clock can be provided upon the development-kit in two ways

4.2.6.3.1. Generate by dividing the internal DVS raw event rate

The internally generated clock exploits the random continuous generation of internal DVS events (The actual generation frequency is fluctuate with the scene). In other word, each clock cycle is generated based on the number of DVS event.

The available counting range is [2^{14}, 2^{17}], the actual *internal_slow_clk_divider* available range is [14, 17]

To set this value:

```
# using internal slow-clk and set the clock counter to 2^17 DVS events.
samna_config.factory_config.internal_slow_clk_divider = 17
```

4.2.6.3.2. Provide by external clock source

On Speck™ development kit, A FPGA is used to provide a stable, programmable slow-clk signal to Speck™. This external clock source is fully independent to the operating of the chip and can be configured via samna.

```
# get the development kit object
devkit = samna.device.open_device("Speck2fModuleDevKit")

# get the io module
devkit_io = devkit.get_io_module()

# enable the slow-clock
devkit_io.set_slow_clock(True)

# set the slow-clock frequency to 1000Hz
devkit_io.set_slow_clk_rate(1000)
```

4.2.6.4. On Board Power Monitoring

The Speck™ development kit has the built-in on board power monitor for five power traces of the chip. These are listed out as **Table 7**:

Table 7: Power Trace of Speck™

Power Trace	Channel	Description
VDD_IO	0	Power of IO interface
VDD_RAM	1	Power of RAM r/w
VDD_LOGIC	2	Power of logic operation
VDD_PIXEL_DIGITAL	3	DVS pixel power from digital circuits

For a simple start up power measurement:

```
import samna
import time

d = samna.device.get_unopened_devices()
dk = samna.device.open_device(d[0])

power = dk.get_power_monitor()
buf = samna.BasicSinkNode_unifirm_ modules_events_monitor()
graph = samna.graph.EventFilterGraph()
graph.sequential([power.get_source_node(), buf])

print("Manual power monitor test:" )
power.single_shot_power_monitor()
time.sleep(1)
ps = buf.get_events()
[print(p) for p in ps]
time.sleep(2)

print("Auto power monitor test:" )
# set freq to 1 Hz. The maximum power monitor rate is 100 Hz
power.start_auto_power_monitor(1.0)
time.sleep(5)
power.stop_auto_power_monitor()
ps = buf.get_events()
[print(p) for p in ps]
```

Note: The on board power monitor has about ± 50uW offset on each power trace. Max sampling rate 100Hz.

4.3. Connecting External DVS Resource

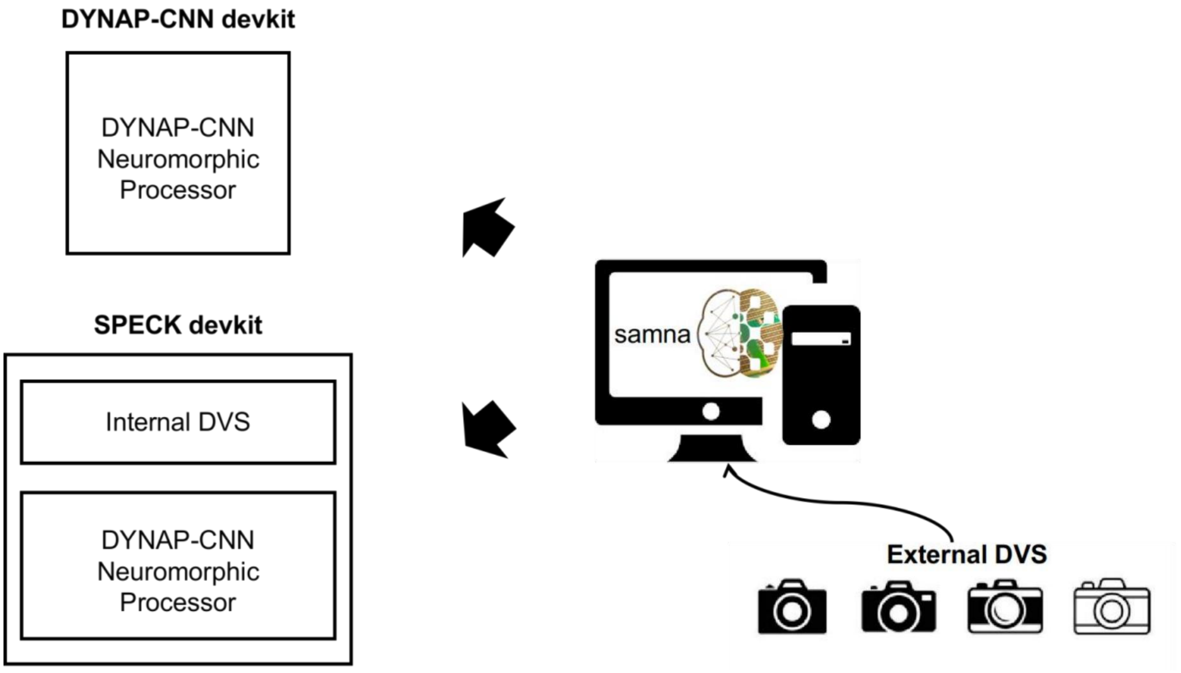

DYNAP-CNN devkit
DYNAP-CNN Neuromorphic Processor
SPECK devkit
Internal DVS
DYNAP-CNN Neuromorphic Processor
samna
External DVS

Figure 1 2 . Diagram Illustration of Connecting External DVS

As is shown in **Figure 11**, It is possible to connect an external DVS camera to the board via host machine, more info can be found at

Send events from a DVS to a dev kit using a graph.

5. Software Tool Chain

SynSense provides Sinabs and Samna to help development on the Speck™ Development Kit.

5.1. Sinabs

Sinabs is a Python library for development and implementation of Spiking Convolutional Neural Networks (SCNNs). The library implements several layers that are spiking equivalents of CNN layers. In addition it provides support to import CNN models implemented in torch conveniently to test their spiking equivalent implementation.

An SNN model developed in Sinabs can be easily deployed onto the Speck™ development kit with the host machine software **Samna**.

Sinabs-dynapcnn is the Plug-in site-package based on Sinabs that support user can create hardware compatible neural networks for Speck™ and DYNAP™CNN chip series. It wraps a number of samna configuration APIs and helps user can do the chip network deployment with few lines of code.

5.2. Samna

Samna is the developer interface to the SynSense tool chain and run-time environment for interacting with all SynSense devices. Developed towards efficiency and user friendly, a set of Python API is available with the core running in C++, it is possible to work with neuromorphic devices in a professional and elegant manner. Samna also features an event based stream filter system allows real-time, multi-branch processing of the event based stream coming in or out from the device. With an integration of a just-in-time compiler in Samna, the flexibility of this filter system has been taken to an even higher dimension, which supports adding users defined filter functions at run-time to meet requirements of any different scenarios.

For more examples please refer to Samna Official Documentation.

To efficiently use the Speck^{TM} Development Kit, it is essential to use the samna graph to build a route that can communicate with dev-kit. The following links provide few instance that assist the user to start with:

Device Controller:

https://synsense.gitlab.io/sinabs-dynapcnn/faqs/device_management.html

Visualization of the DVS:

https://synsense-sys-int.gitlab.io/samna/devkits/speckSeries/examples/display_speck2f_dvs.html

Measurement of Power：

https://synsense.gitlab.io/sinabs-dynapcnn/getting_started/notebooks/power_monitoring.html

Speck Event API List：

https://synsense-sys-int.gitlab.io/samna/reference/speck2f/event/index.htmlx

Speck Configuration API List：

https://synsense-sys-int.gitlab.io/samna/0.48.0/reference/speck2f/configuration/index.html#samna.speck2f.configuration.SpeckConfiguration

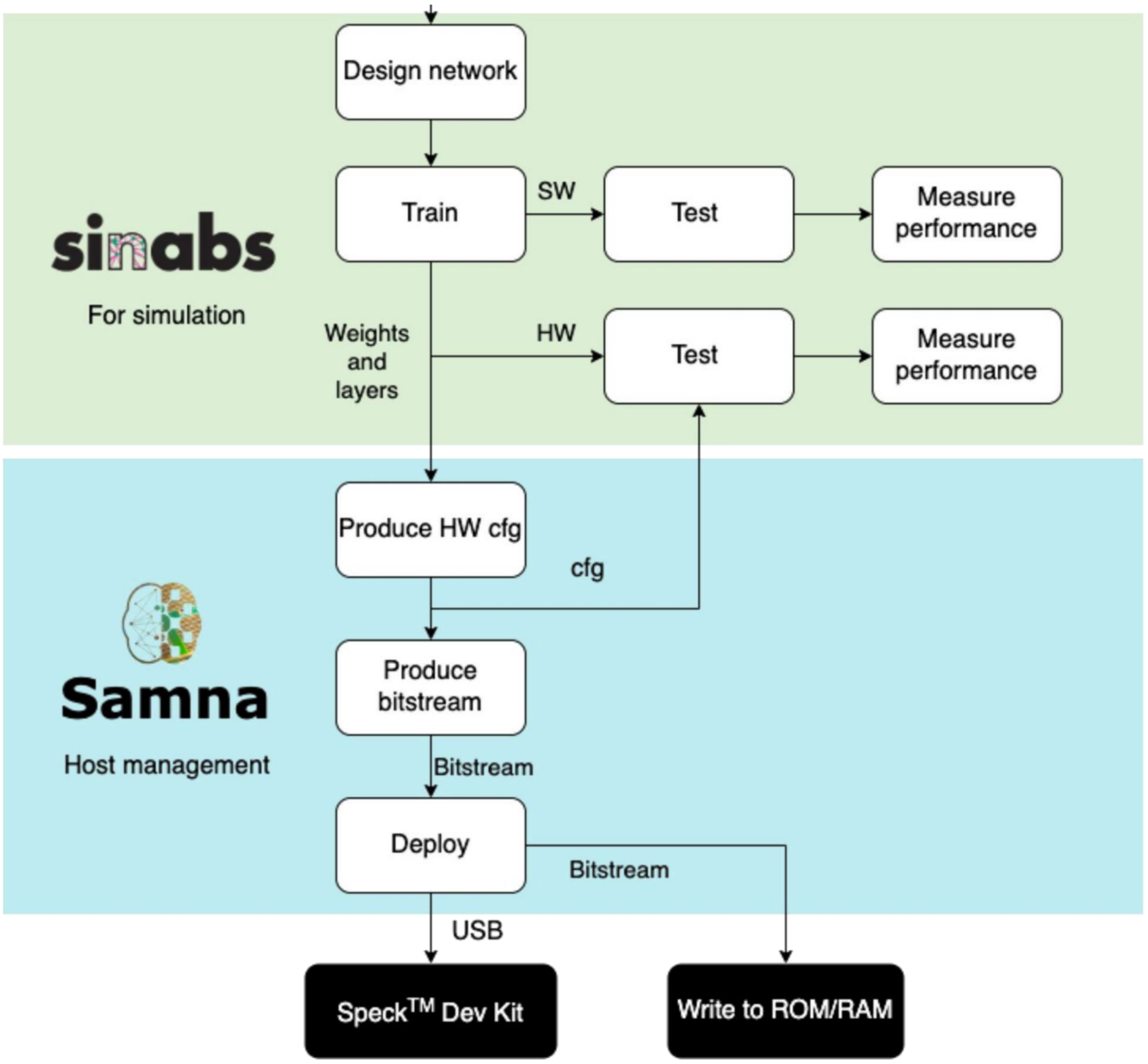


Figure 1 3 . Software Tool Chain for Speck™

5.4. DVS tools

DVS tool mainly focusing on provide the user ability to record and label the data using Speck™ development kit. The tools is maintained at git repository:

https://gitlab.com/synsense/dvs_tool

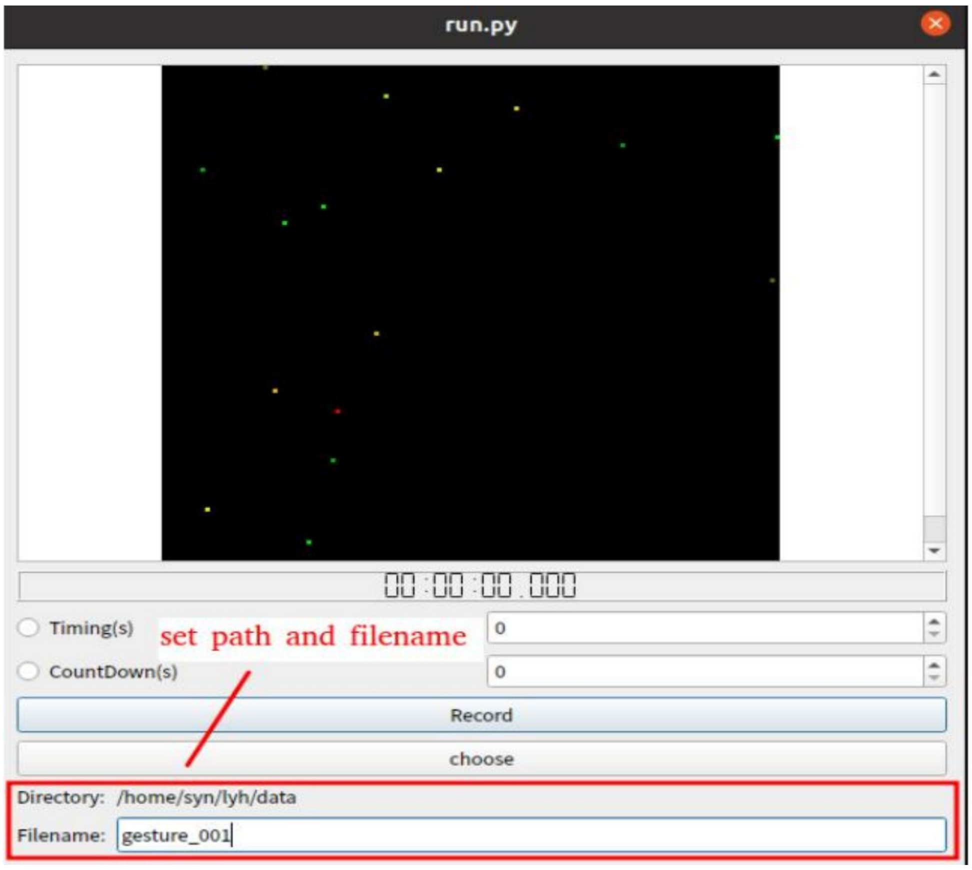
Note: Recorder and Labeling tool are two software that maintained in different branch, please download the tools separately from branch instead of clone the entire repository.

5.4.1 DVS Recorder

With Python terminal:

```
pip install -r requirements.txt
python run.py
```

1. Start to record Click the Record menu and click record, then choose a device(if there are many devices)



Timing(s): 0
CountDown(s): 0
Directory: /home/syn/lyh/data
Filename: gesture_001

Figure 1 4 . Set path for DVS recorder

2. Set storage path and filename as is shown in Figure 13.

3. Start and stop recording, with the adjustment of record timing and countdown as shown in Figure 14.

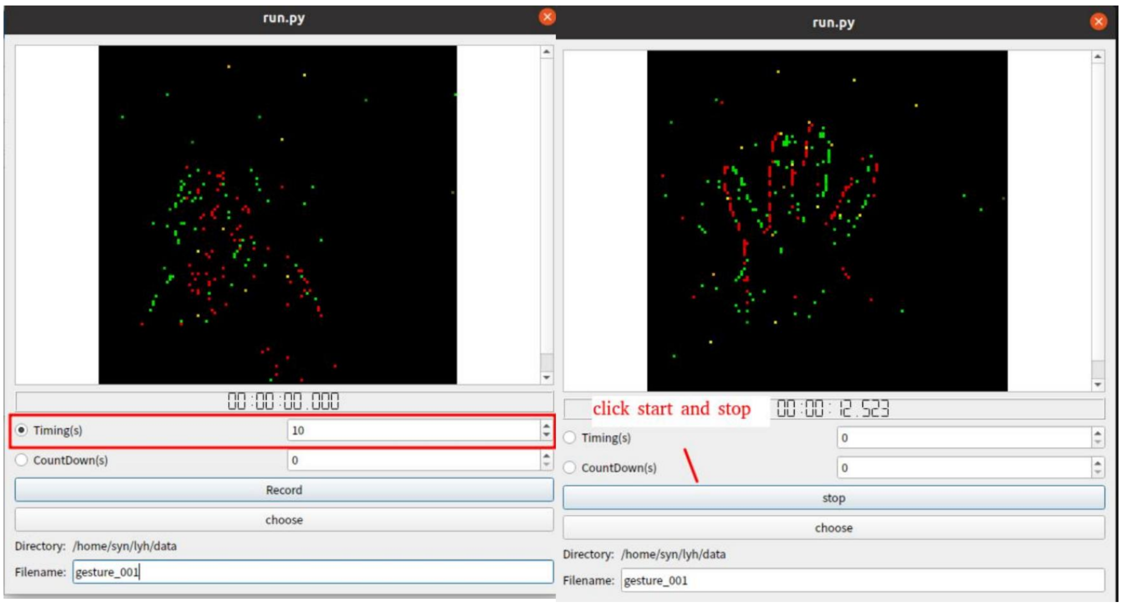


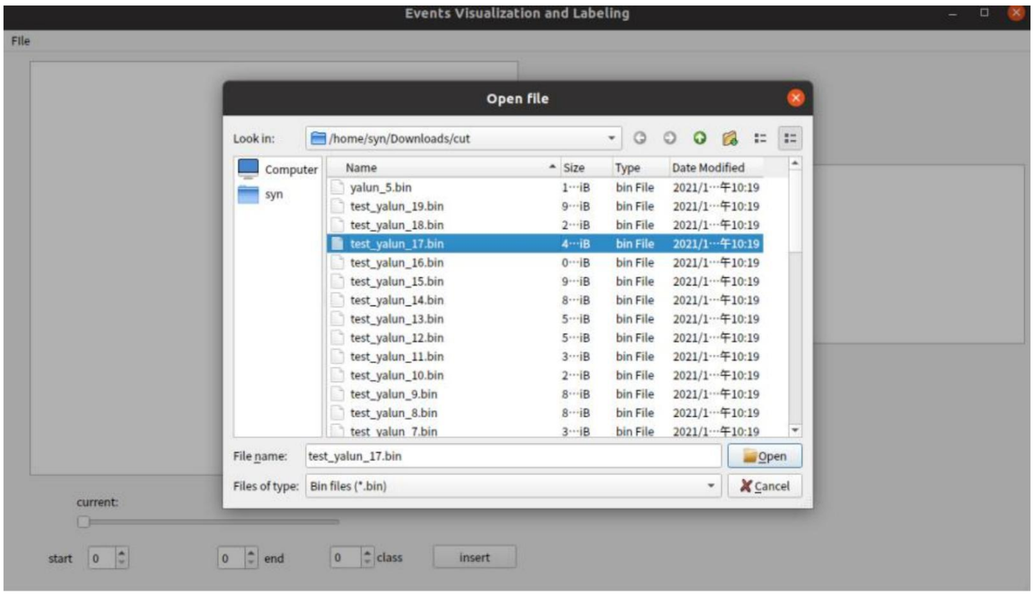
Figure 1 5 . DVS tool record and settings

5.4.2 DVS Labeling Tool

With Python terminal:

```
pip install -r requirements.txt
python run.py
```

1. Click the File menu then choose a data file (*bin), as shown in Figure 15.

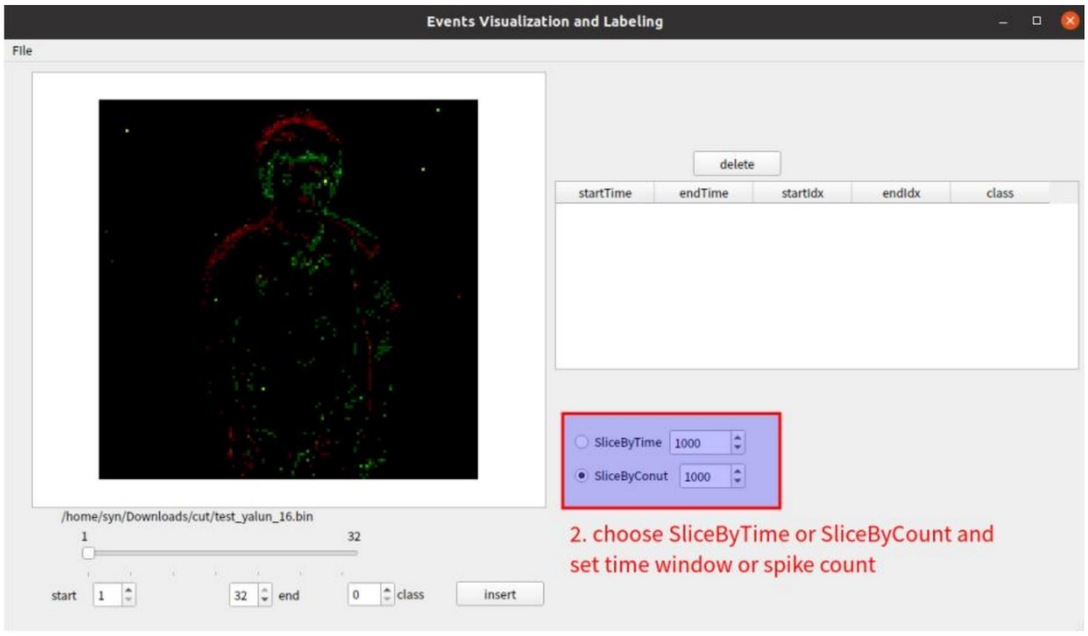

Events Visualization and Labeling
File
Open file
Look in: /home/syn/Downloads/cut
Computer
syn
Open
Cancel
current:
start
end
class
insert

Name	Size	Type	Date Modified
yalun_5.bin	1···iB	bin File	2021/1···午10:19
test_yalun_19.bin	9···iB	bin File	2021/1···午10:19
test_yalun_18.bin	2···iB	bin File	2021/1···午10:19
test_yalun_17.bin	4···iB	bin File	2021/1···午10:19
test_yalun_16.bin	0···iB	bin File	2021/1···午10:19
test_yalun_15.bin	9···iB	bin File	2021/1···午10:19
test_yalun_14.bin	8···iB	bin File	2021/1···午10:19
test_yalun_13.bin	5···iB	bin File	2021/1···午10:19
test_yalun_12.bin	5···iB	bin File	2021/1···午10:19
test_yalun_11.bin	3···iB	bin File	2021/1···午10:19
test_yalun_10.bin	2···iB	bin File	2021/1···午10:19
test_yalun_9.bin	8···iB	bin File	2021/1···午10:19
test_yalun_8.bin	8···iB	bin File	2021/1···午10:19
test_yalun_7.bin	3···iB	bin File	2021/1···午10:19

File name: test_yalun_17.bin
Files of type: Bin files (*.bin)

Figure 1 6 . DVS tool open bin file

2. Choose slicing method to set framing visualization performance(Figure 16).

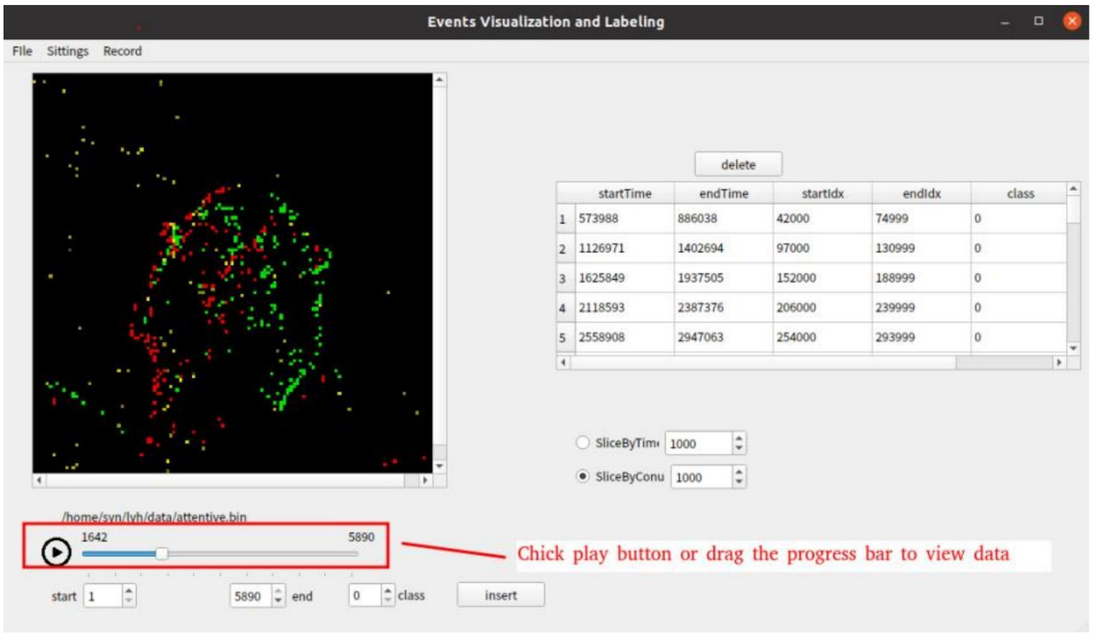

Events Visualization and Labeling
File
delete
startTime
endTime
startIdx
endIdx
class
/home/syn/Downloads/cut/test_yalun_16.bin
1
32
start
end
class
insert

○ SliceByTime 1000
● SliceByConut 1000
start 1 | 32 end | 0 class

2. choose SliceByTime or SliceByCount and set time window or spike count

Figure 1 7 . DVS tool slicing setting

3. Drag the progress bar or click the play button to visualize the data(Figure 17).


Events Visualization and Labeling
File
Sittings
Record
delete
/home/syn/lvh/data/attentive.bin
1642
5890
start
end
class
insert

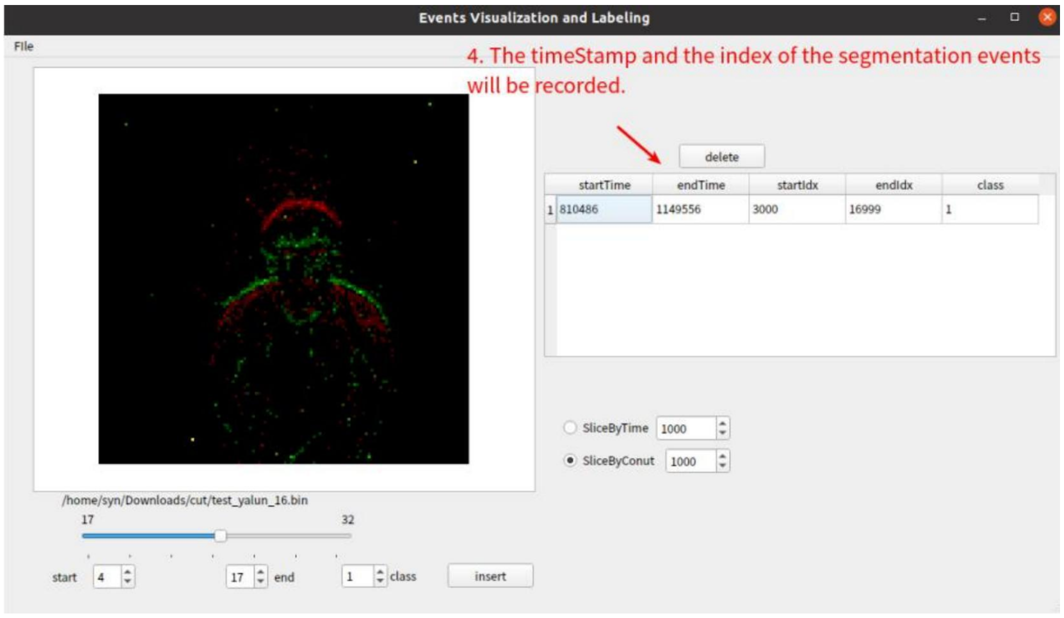
	startTime	endTime	startIdx	endIdx	class
1	573988	886038	42000	74999	0
2	1126971	1402694	97000	130999	0
3	1625849	1937505	152000	188999	0
4	2118593	2387376	206000	239999	0
5	2558908	2947063	254000	293999	0

○ SliceByTim 1000
● SliceByConu 1000
start 1 | 5890 end | 0 class

Chick play button or drag the progress bar to view data

Figure 1 8 . DVS tool visualization

4. Click insert to add info of segmentation(Figure 18) and double click label visualize the labeled data(Figure 19).



4. The timeStamp and the index of the segmentation events will be recorded.

	startTime	endTime	startIdx	endIdx	class
1	810486	1149556	3000	16999	1

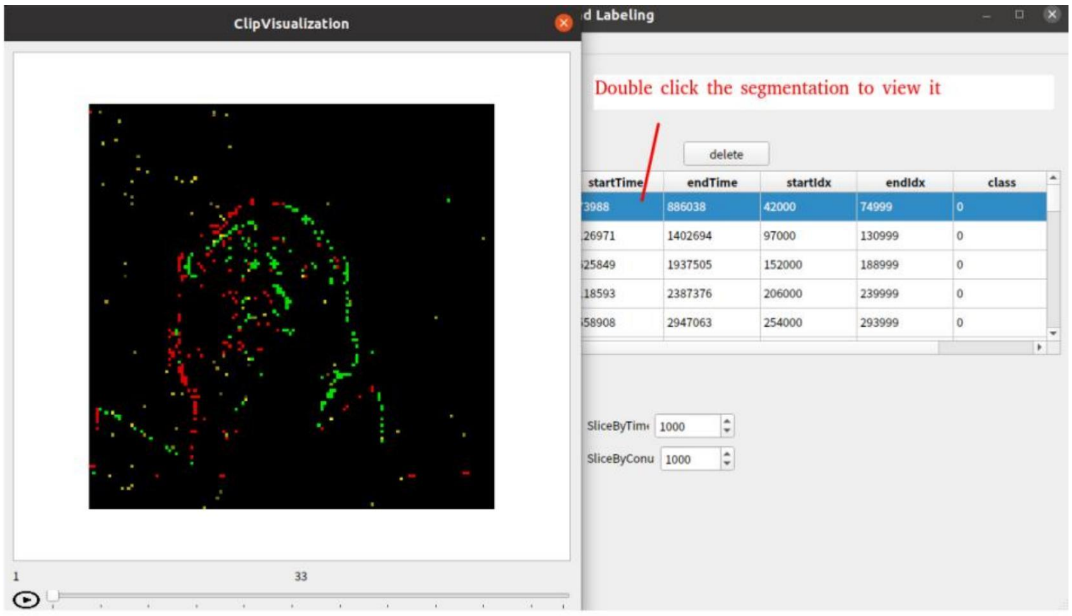
( ) SliceByTime 1000
(●) SliceByConut 1000

/home/syn/Downloads/cut/test_yalun_16.bin

17 32

start 4 17 end 1 class

Figure 1 9 . DVS tool add label



Double click the segmentation to view it

startTime	endTime	startIdx	endIdx	class
3988	886038	42000	74999	0
26971	1402694	97000	130999	0
25849	1937505	152000	188999	0
18593	2387376	206000	239999	0
58908	2947063	254000	293999	0

SliceByTim 1000
SliceByConu 1000

Figure 2 0 . DVS tool visualize the labeled data

6. Technical Support

For a more detailed explanation of the reading principle and method for the Speck™ chip output, as well as instructions for configuring and utilizing the on-chip CNN and other resources, please visit SynSense's publicly available materials on:

GITHUB:

https://github.com/synsense

Sinabs Documentation:

https://sinabs.ai/

Git Repository:

https://github.com/synsense/sinabs

Sinabs-dynapcnn Documentation:

https://synsense.gitlab.io/sinabs-dynapcnn/

Samna Documentation:

https://synsense-sys-int.gitlab.io/samna/

For further inquiries please visit:

https://www.synsense.ai/contact/

7. Change log

No.	Version	Date	Editor	Changes
1	V0.1	2023.04	SI	Initial Version
2	V1.0	2023.08	AL	1st Version
3	V1.1	2024.11	Dylan	Added getting started part
4	V1.2	2025.12	JZ	Updated links

Make Intelligence smarter

© SynSense — All rights reserved