Lab manual
SpiNNaker interfacing external devices

1 – Introduction

This portion of the manual introduces the interfaces between the SpiNNaker system and the real world, using both sensors (retinas, cochleas, etc.) and actuators (motor controllers, etc.).

2 – Installation of the plugin

The I/O interfaces are provided by the sPyNNakerExternalDevicesPlugin module, which can be installed using the command

```
sudo pip install sPyNNakerExternalDevicesPlugin
```

for a system-wide installation, or for a different method (e.g. virtualenv), please refer to sPyNNaker module installation instructions, adapting the commands to the sPyNNakerExternalDevicesPlugin module.

3 – EIEIO Protocol

The EIEIO protocol, acronym for “External/Internal Event Input/Output”, is used to communicate spike events between neuromorphic devices and simulators. The protocol is transport-independent, and it may be used over a wide variety of communication channels. Each packet consists of a 16-bit packet header followed by data. The communication is stateless, meaning that all the information required to interpret the data is contained in the header itself. The format of the header is:

Bit															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
P	F	D	T	Type		Version		Count							

The decode of the header configuration bits is as follows:

Field name	**Position**	**Value**	**Description**
P&F	15 - 14	00	Basic data packet, no key prefix
		01	Command packet
		10	Data packet with lower halfword key prefix
		11	Data packet with upper halfword key prefix
D	13	0	No payload prefix
		1	With payload prefix
T	12	0	Payload are not timestamps
		1	Payload are timestamps
Type	11 - 10	00	16-bit key
		01	16-bit key and payload, alternating
		10	32-bit key
		11	32-bit key and payload, alternating
Version	9 - 8	00	Version 0 of the protocol

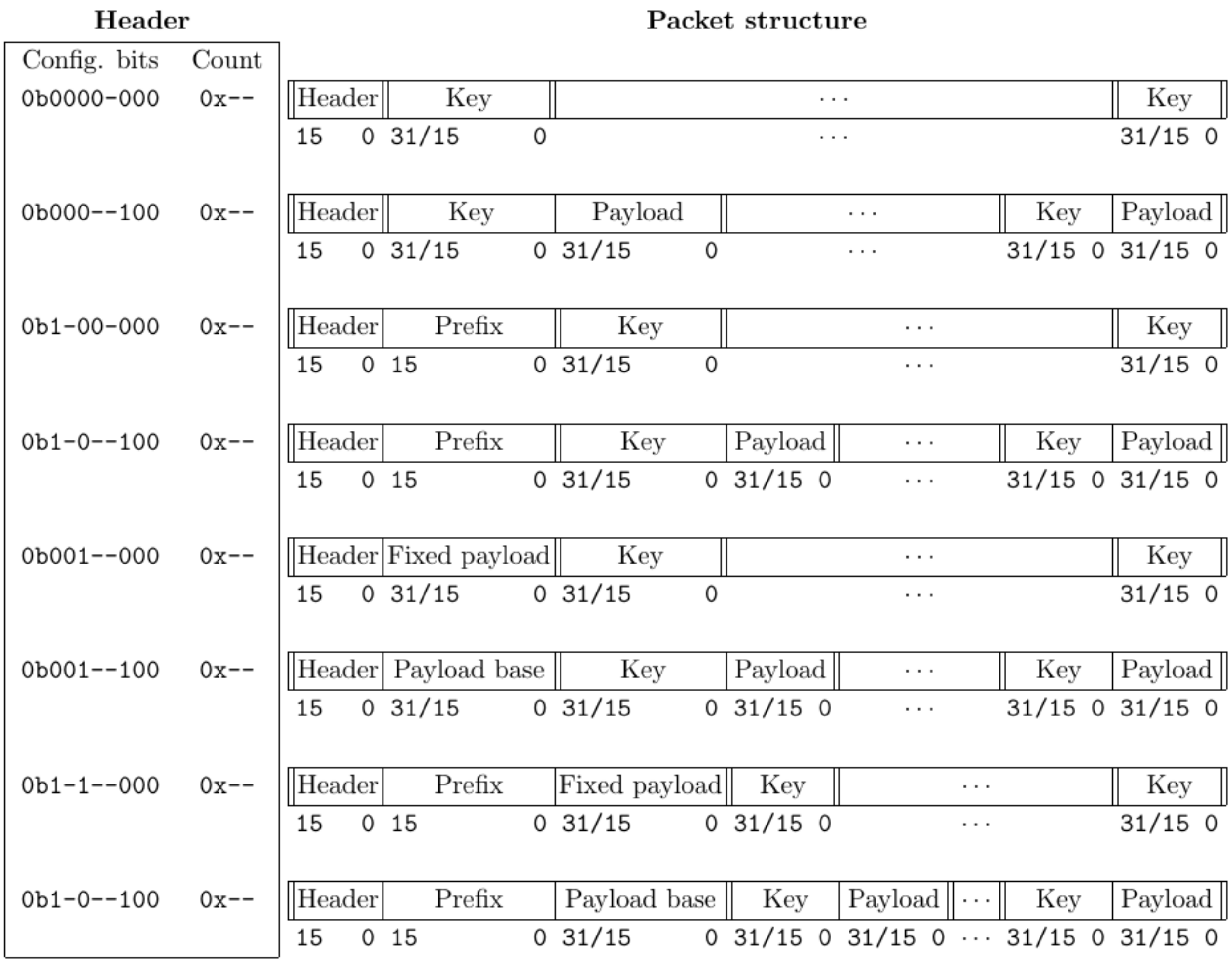
The count parameter indicates how many entries there are in the packet. The maximum value is 256, however, any device may support only packets up to a certain size. Any transmission needs to be limited to the smallest of the supported sizes between the sender and the receiver. Any prefix signalled in the packet header, either related to the key or to the payload, needs to be OR-ed with the value in the data part of the packet to obtain the transmitted value.

The header offers the possibility to send 9 types of EIEIO packets: one command packet format and 8 data type formats. For the command packet, the bits are organized as follows:

Bit			
15	**14**	**13-0**	**Payload**
0	1	Command ID	Command and device-specific

Where the command ID is a 14-bit number. The identifier of the command and the packet payload are specific of the device and the command.

The data packet formats and the correspondent header configuration bits are:



Numbers starting with prefix '0b' are expressed in binary format. Numbers starting with prefix '0x' are expressed in hexadecimal format.

The numbers below the packet template indicate the bit positioning in the packet. Multi-byte values are transmitted following the little-endian ordering of the bytes. The size of any portion of the packet is defined by “type” bits (bits 11-10 in the header), which have been left undefined in the header configuration bits.

Where the prefix is included, the bit related to the upper or lower half word prefix (bit 14) is left undefined, as the format of the packet is unaffected. Where the packet presents payload, the timestamp bit (bit 12) is left undefined, as the format of the packet is unaffected. The number of keys (or couples key/payload) are defined by the “Count” parameter in the header field.

The optional field “prefix” is used to reduce the amount of data which is transmitted with each event in a packet. If all the events have e.g. a common upper half-word, then this can be indicated at transmission time in the packet as a prefix. The receiver, parsing the packet, will reconstruct the key performing an OR operation: the prefix is shifted in the correct position (e.g. upper or lower half-word) and each key contained in the packet is OR-ed with such prefix. The same infrastructure is in place for payload, with one addition: it is possible for a packet to identify only one payload which is common for all the key transmitted with a packet. The payload may also be used to identify the timestamp of each event. The common payload may useful e.g. to identify the time at which the events contained in a single packet were generated.

4 – Models for external interfaces

External interfaces can be subdivided in two classes: input models, which sense the environment and provide spikes to the SpiNNaker system, and output models, which receive spikes from SpiNNaker and perform operations in the real world, such as moving a robot.

SpiNNaker supports input injection using three different methods: Ethernet interface, spiNNaker link and spiNNlink. The first method is relatively slow, but uses a "plug-and-play" network protocol, UDP, to transport spikes encoded in EIEIO packets (EIEIO over UDP).

SpiNNaker link uses a 2-of-7 encoding to transmit packet directly to the SpiNNaker router. All the details for this protocol are specified in the "Application Note 7 – SpiNNaker Links" [Steve Temple, 2012].

Finally, although spiNNlinks use SATA connectors and cabling, the protocol used for the transmission allows to funnel 8 spiNNaker links into a single spiNNlink.

In general input to be provided may be classified in four possible classes:

1. Input is known before the start of the simulation and it may be stored entirely in simulator's memory.
2. Input is known before the start of the simulation and it cannot be stored in its entirety in simulator's memory.
3. Input is unknown before the start of the simulation and information refers to the current time of the simulation.
4. Input is unknown before the start of the simulation and information received may refer to a different time (e.g. future) of the simulation.

In PyNN, SpikeSourceArray populations may classify either in the first or the second class, depending on the size of the spike train to be injected.

Input/Output devices, instead, generate input which belongs to class 3 (generally speaking – e.g. retinas on a robot feeding a neural network to allow a robot to move in the environment). Such models require the description of the connector to which each device connects. The description of the connector is an identifier starting from 0 to n-1, where n is the number of spiNNaker link connectors on the board.

As indication for versions 3 and 5 of the SpiNNaker boards, following are two images to represent connector IDs.

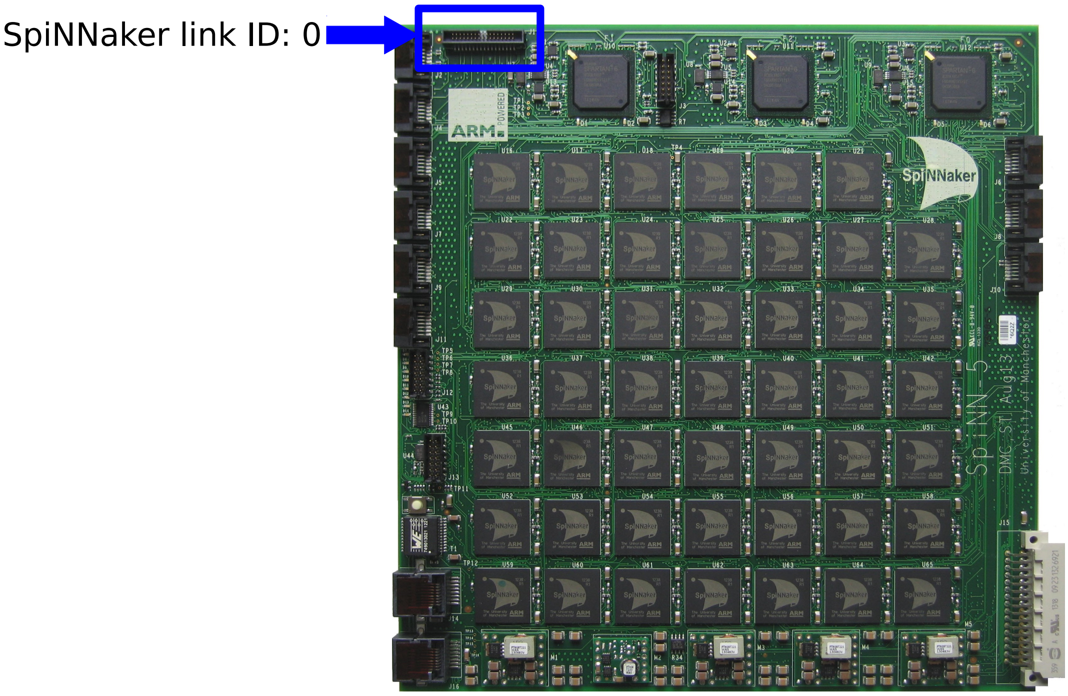


Illustration 1: SpiNNaker version 5 board

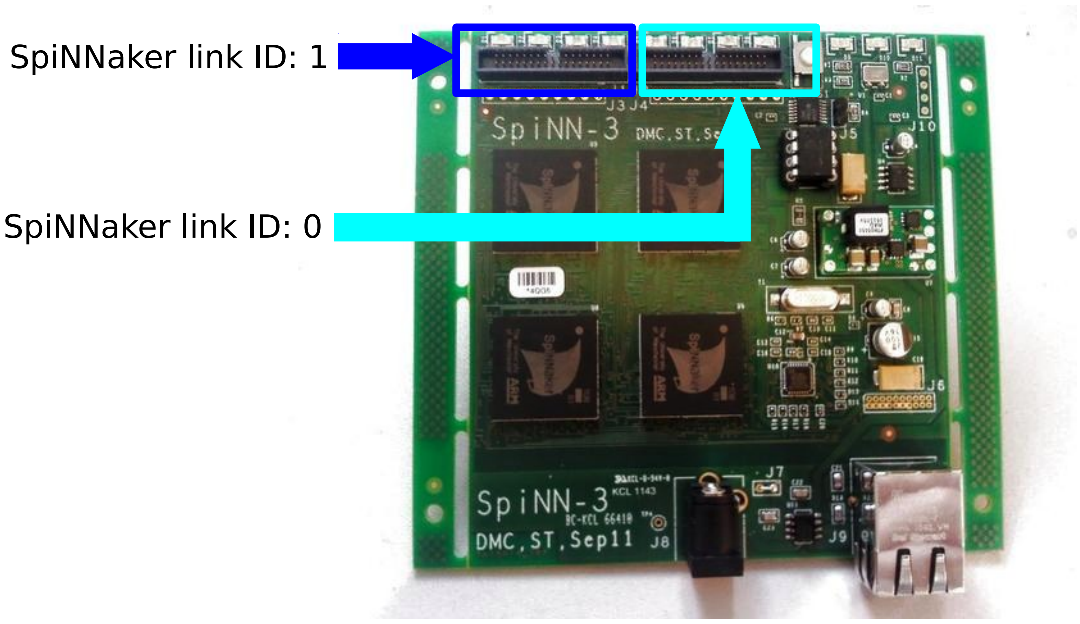


Illustration 2: SpiNNaker version 3 board

The following models are defined to be used in PyNN scripts to define the I/O interfaces

4.1 – Input model

This set of models includes examples of PyNN population models used to input and output spikes to/from SpiNNaker using one of the interfaces described above.

4.1.1 – Spike Injector

This is the most common of the interfaces as it provides a way to inject spikes using the Ethernet interfaces. This is mainly described in the simple I/O lab manual, together with a number of examples. However, here we cover the same topic in more details.

The Spike Injector model was originally designed to propagate spikes coming from the Ethernet channel, at the time they are received. Subsequently, there buffering functionality has been added, so that future spikes may be stored for future use.

The sum of these features covers the requirement for the SpikeSourceArray PyNN population, covering both classes 1 and 2 of input described earlier. However, also classes 3 and 4 are covered, since the input may be provided either from a host PC or an external device communicating with EIEIO protocol.

The only restriction is that the input provided during the simulation need to refer to monotonically increasing timestamps, as the simulation moves only forward in time.

EIEIO packets identified by the on-board population include all the types of packet specified by the protocol. In case packets specify 32-bit neural events, the on-board population propagates multicast packets with the specified 32-bit routing keys, without modifications. However, in case received packets include only 16 bit keys (with or without payload), these need to be extended to 32 bits, as the SpiNNaker architecture is 32-bit wide. If the packet specifies a key prefix, this is added as specified in the packet (upper/lower halfword) and the key is transmitted in multicast packet without further modifications.

If the key is 16 bit wide, with no prefix, The application on SpiNNaker automatically converts the 16-bit key to 32 bits prepending the source population ID.

4.1.2 – Cochlea Device

This device represents a silicon cochlea which uses an FPGA to interface the spiNNaker link on the SpiNNaker hardware. A PyNN population using this model requires one parameter to be configured correctly:

```
parameters = {
    'spinnaker_link': [0 or 1]
    }
```

The parameter spinnaker_link represents the link on which the interface FPGA is connected, as described previously.

4.1.3 – FPGA Retina Device

This device represents a silicon retina which uses an FPGA to interface the spiNNaker link on the SpiNNaker hardware. The retina produces spikes for an area comprising 128 x 128 elements. A PyNN population using this model requires a few parameters to be configured correctly:

```
parameters = {
    'mode': [MODE_128, MODE_64, MODE_32 or MODE_16],
    'retina_key': [16 bit value],
    'spinnaker_link_id': [0 or 1],
    'polarity':
    }
```

The interface FPGA may be configured to subsample the input spikes from 128 x 128 to 64 x 64, 32 x 32 or 16 x 16, condensing all the spikes received to a smaller number of picture elements. This is specified also in the parameters of the model with the 'mode' parameter. The value which may be assigned to this parameter are specified in the ExternalFPGARetinaDevice class and are:

`ExternalFPGARetinaDevice.MODE_128`	retina input at resolution 128 x 128 (full resolution)
`ExternalFPGARetinaDevice.MODE_64`	retina input at resolution 64 x 64
`ExternalFPGARetinaDevice.MODE_32`	retina input at resolution 32 x 32
`ExternalFPGARetinaDevice.MODE_16`	retina input at resolution 16 x 16

Parameter 'retina_key' identifies the top 16 bits of the routing key arriving from the FPGA interface.

The parameter spinnaker_link represents the link on which the interface FPGA is connected, as described previously.

Polarity identifies the polarity of the spikes that the population represents. The definition of polarity is described by the retina documentation and identifies if the change in light reported by the spike event is an increase (positive polarity) or decrease (negative polarity). The values this parameter can assume are specified in the ExternalFPGARetinaDevice class and are:

`ExternalFPGARetinaDevice.UP_POLARITY`	retina input from the positive polarity field
`ExternalFPGARetinaDevice.DOWN_POLARITY`	retina input from the negative polarity field
`ExternalFPGARetinaDevice.MERGED_POLARITY`	retina input from both the positive and the negative polarity fields

4.2 – Output models

This set of models allow conversion of neural events from SpiNNaker format into different formats which may by used interfaces outside the SpiNNaker simulator. The following is an example of output interface. However more output interface are available for specific tasks, e.g. robot wheel control.

4.2.1 – Live Packet Gatherer

This model receives multicast packets from the SpiNNaker network and groups them into EIEIO packets to be sent to the output world via Ethernet.

This model can be instantiated in PyNN using the activate_live_output_for() API call. This allows to set up a streaming channel to the external world. The EIEIO packets sent out are set by default to include 32 bit keys and a fixed payload which represents the timestamp in which the keys have been generated. The packet format may be defined by the user configuring the appropriate optional parameters of the function activate_live_output_for(). A summary list of the available parameters with the default value and a description is:

Parameter	**Default**	**Description**
`population`	-	PyNN population that will send spikes to the output world
`port`	As per configuration file spynnaker.cfg	Target UDP port for the packets sent
`host`	As per configuration file spynnaker.cfg	Target host for the packets sent
`use_prefix`	False	Use a key prefix in packet header
`key_prefix`	None	Key prefix to use
`prefix_type`	None	Upper / Lower halfword for key prefix
`message_type`	32 bit without payload	16 / 32 bit keys and payloads

right_shift	0	If keys to be sent are 16 bits, 32-bit keys from SpiNNaker may be right-shifted
payload_as_time_stamps	True	Payload indicate time stamp of events
use_payload_prefix	True	Use a payload prefix in packet header
payload_prefix	None	Payload prefix to use

The content of the packet (parameter message_type) is specified using a few constants defined in the enum EIEIOType define in module spinnman.messages.eieio.eieio_type . Possible values are described in the table below:

EIEIOType.KEY_16_BIT	Indicates that data in the EIEIO packet is keys which are 16 bits
EIEIOType.KEY_PAYLOAD_16_BIT	Indicates that data in the EIEIO packet is keys and payloads of 16 bits
EIEIOType.KEY_32_BIT	Indicates that data in the EIEIO packet is keys of 32 bits
EIEIOType.KEY_PAYLOAD_32_BIT	Indicates that data in the EIEIO packet is keys and payloads of 32 bits

5 – Live Connection Handler

This class is instantiated in PyNN on the host and it is used to handle streaming connections to/from SpiNNaker. The class itself allows to select a listening interface and UDP port ro receive spikes and to associate the population label(s) to which and from which events are communicated.

The initialization of this class allows a few parameters for configuration:

Parameter	**Default**	**Description**
receive_labels	None	Label(s) of the population(s) from which spikes are received
send_labels	None	Label(s) of the population(s) to which spikes will be sent
local_port	19999	(optional) UDP port on which to listen for incoming messages
local_host	None	IP address of the interface to listen for incoming messages

And the operations allowed by this class are described in the following table:

Function	Description
`add_receive_callback`	Add a function call upon receiving a message from SpiNNaker
`add_start_callback`	Add a function call upon start of the simulation
`send_spike`	Send an EIEIO packet with a single key to SpiNNaker
`send_spikes`	Send an EIEIO packet with multiple keys to SpiNNaker

The first two functions can be used to generate a callback to a user-defined function upon particular events: a message being received from SpiNNaker (the first function), or the start of the simulation on SpiNNaker (the second function).

The function added with the `add_receive_callback` receives as parameters the identifier of the neuron within the population together with the label of the population. The prototype of this function needs to be:

```
def receive_callback(label, time, neuron_ids)
```

Where the parameters used in the call are:

Parameter	Description
`label`	Label of the population which generated the set of spikes
`time`	Time at which the spikes have been generated
`neuron_ids`	List of neuron IDs contained in the received packets

The function added with the add_start_callback receives as parameters the label of the population which is going to receive spikes and the live connection object that needs to be used to send spikes

```
def start_sender(label, sender)
```

The parameters used in the call are:

Parameter	Description
`label`	Label of the population which generated the set of spikes
`sender`	Live connection object to use to send spikes to the SpiNNaker machine

The functions “send_spike” and “send_spikes” are used to send respectively one spike or a list of spikes to the SpiNNaker machine. The parameters to use for these function calls are:

Parameter	Description
`label`	Label of the population which is going to inject the spike(s)
`neuron_id`	ID(s) of the neuron(s) which are to inject the spike(s) in the simulation. Single value for function “send_spike”, list of values for function “send_spikes”

7 – Tasks with external interfaces

Task 1 – Injected Packets from an External Device [Easy]

The idea of this task is to simulate the injection of packets from a more realistic hardware device i.e. one that cannot read the database. In order to simulate this device, you will need to open a socket to talk to the machine. As this lab is not about creating sockets, some example code is provided to help with this part of the task. This is available from here:

```
http://spinnakermanchester.github.io/2015.005-
Arbitrary/examples/advanced_external_devices_1.py
```

To run the device, run:

```
python advanced_external_devices_1.py <machine-name> <port> <key>
```

where `<machine-name>` is the IP address of your SpiNNaker board, `<key>` is the base key to use to send spikes, and `<port>` is the port that the injector is set up to listen on.

The task, then, is to set up injection from this device. The device transmits spikes for 10 neurons using a fixed base key specified on the command line. Set up a SpikeInjector population to receive the spikes and a connect this to an `IF_curr_exp` population with 10 neurons, using a 1-to-1 connection with weights of 5nA. Record the `IF_curr_exp` population and create a graph of spikes.

The script should be run for approximately 6000 milliseconds.

Task 2: Sending Packets to an External Device [Moderate]

The idea of this task is to simulate the reception of packets at an external device. Again, a device has been created for you here:

```
http://spinnakermanchester.github.io/2015.005-
Arbitrary/examples/advanced_external_devices_2.py
```

To run the device, run:

```
python advanced_external_devices_2.py <port>
```

where <port> is the port that the spikes are being sent to

The task is to set up the sending of spikes to this device. The device has 4 neurons, for forwards, backwards, left and right. Set up a spike source array and activate live output for the device. The device expects to receive only 16-bit keys, and doesn't understand any other packet formats i.e. no prefix or timestamp is expected. Send some spikes from the spike source array and look at the output from the script.

Task 3: SpiNNakerLink device [Hard]

The idea of this task is to go through the motions of creating a device that connects to a SpiNNaker link. Although you won't actually have such a device, the software can still set up connections as if this were the case, and you can then examine the routing tables and trace where the packets will go. A script has been created to read the routing tables. This is available here:

`http://spinnakermanchester.github.io/2015.005-Arbitrary/examples/advanced_external_devices_3.py`

To run the device, run:

```
python advanced_external_devices_3.py <machine_name> <x> <y>
```

where `<machine-name>` is the IP address of your SpiNNaker board, `<x>` is the x-coordinate of a chip on the board, and `<y>` is the y-coordinate of a chip on the board.

The task is to create a virtual device vertex that can be used as a Population in a PyNN script. This is done as follows:

1. Create a new class which extends from `pacman.model.abstract_classes.abstract_virtual_vertex.AbstractVirtualVertex` and also from `spinn_front_end_common.abstract_models.abstract_outgoing_edge_same_contiguous_keys_restrictor.AbstractOutgoingEdgeSameContiguousKeysRestrictor`
2. The initializer of the class need to take the parameters to be compatible with the PyNN interface: `machine_time_step, timescale_factor, spikes_per_second, ring_buffer_sigma, label, n_neurons, constraints`
3. The initializer also needs to take a `spinnaker_link_id` parameter to pass on to the `AbstractVirtualVertex`
4. Check that the user value of `n_neurons` is compatible with your device.
5. Call the `AbstractVirtualVertex` initializer, passing the number of neurons (`n_neurons`), the `label` and the `max_atoms_per_core`, which should be equal to the number of neurons (to avoid the device being partitioned).

6. Call the `AbstractOutgoingEdgeSameContiguousKeysRestrictor` initializer.
7. Add a method
   `get_outgoing_edge_constraints(self, partitioned_edge, graph_mapper)`
   that will get the constraints from the
   `AbstractOutgoingEdgeSameContiguousKeysRestrictor`
   and then append to this list an additional constraint:
   `KeyAllocatorFixedKeyAndMaskConstraint([KeyAndMask(0x42000000, 0xFFFF0000)])`
   (assuming you have stored the key and mask using these variable names).
8. Add method `is_virtual_vertex`, which should return `True`
9. Add method `get_model_name` which should return a string containing the name of your device

Create a PyNN script which creates a population of your device using a spinnaker_link_id of 0, and choose a key and mask for your device. Create a population of neurons and a projection from the device population to the population of neurons. Run the simulation for 10ms (it is not critical, as the device isn’t really there).

Using the script downloaded above, get the routing table for chip 0, 0. Look at the entries and see if you can see those for your device.