

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:

Header		Packet structure														
Config. bits	Count															
0b0000-000	0x--	Header	Key		...					Key						
		15	0	31/15	0	...					31/15	0				
0b000--100	0x--	Header	Key		Payload		...			Key	Payload					
		15	0	31/15	0	31/15	0	...			31/15	0	31/15	0		
0b1-00-000	0x--	Header	Prefix		Key		...			Key						
		15	0	15	0	31/15	0	...			31/15	0				
0b1-0--100	0x--	Header	Prefix		Key	Payload	...			Key	Payload					
		15	0	15	0	31/15	0	31/15	0	...			31/15	0	31/15	0
0b001--000	0x--	Header	Fixed payload		Key		...			Key						
		15	0	31/15	0	31/15	0	...			31/15	0				
0b001--100	0x--	Header	Payload base		Key	Payload	...			Key	Payload					
		15	0	31/15	0	31/15	0	31/15	0	...			31/15	0	31/15	0
0b1-1--000	0x--	Header	Prefix		Fixed payload		Key	...			Key					
		15	0	15	0	31/15	0	31/15	0	...			31/15	0		
0b1-0--100	0x--	Header	Prefix		Payload base		Key	Payload	...	Key	Payload					
		15	0	15	0	31/15	0	31/15	0	31/15	0	...	31/15	0	31/15	0

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 keys transmitted with a packet. The payload may also be used to identify the timestamp of each event. The common payload may be 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 packets 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 an indication for versions 3 and 5 of the SpiNNaker boards, following are two

images to represent connector IDs.

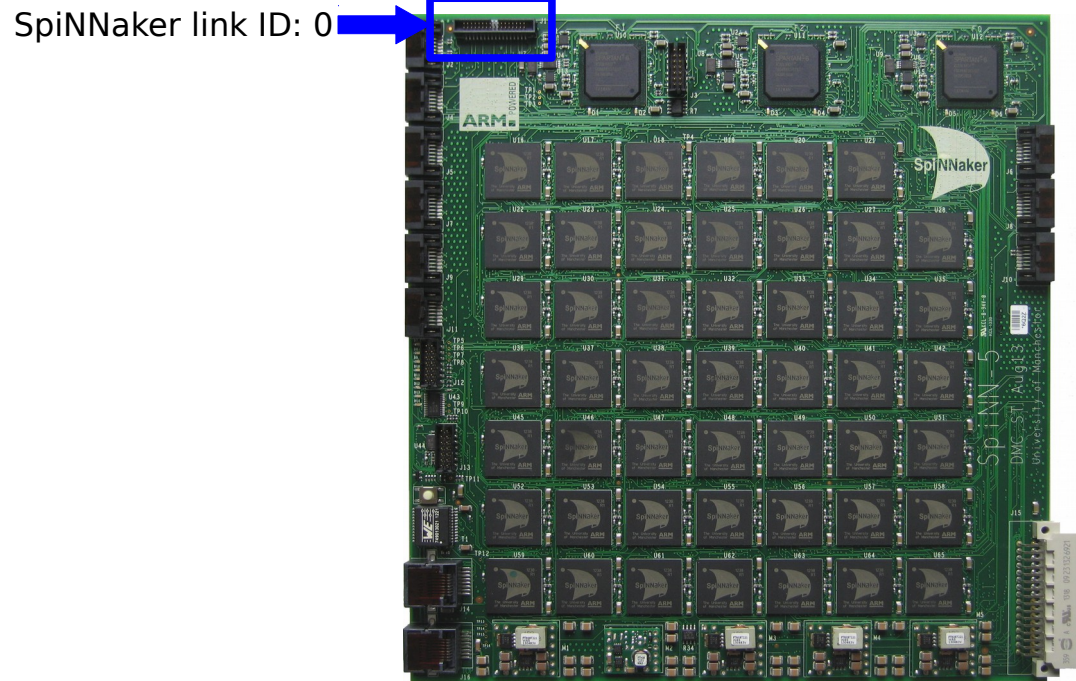


Illustration 1: SpiNNaker version 5 board

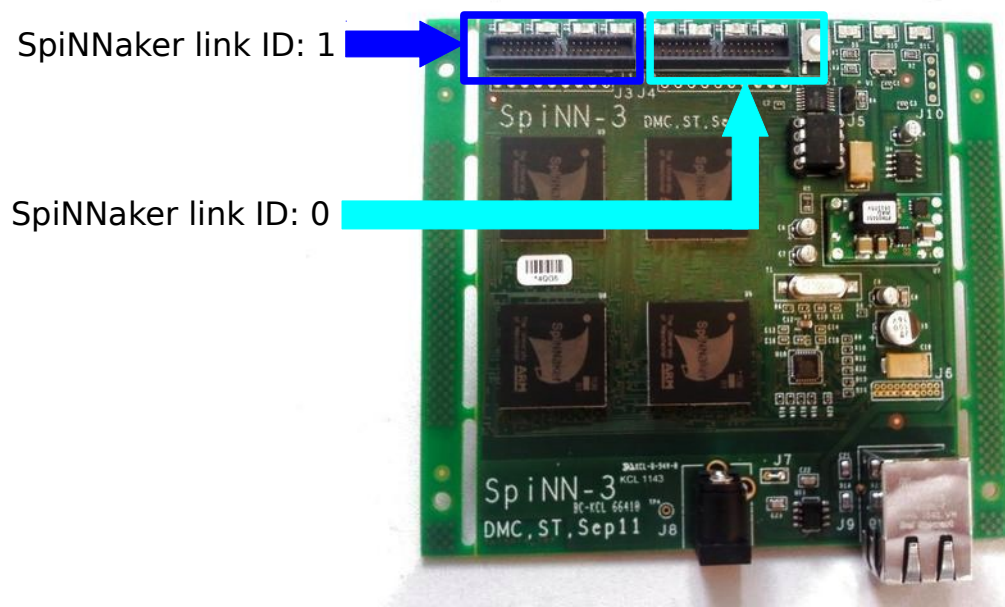


Illustration 2: SpiNNaker version 3 board

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 - Input From Ethernet

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 SpikeInjector 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.

4.2 - Output via Ethernet

The `activate_live_output_for` function is used to activate the live output of multicast packets from the SpiNNaker network. The multicast packets are grouped into EIEIO packets to be sent to the output world via Ethernet. 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. A summary list of the available parameters with the default value and a description is shown below:

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:

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

4.3 - SpiNNakerLink devices

Devices connected to a SpiNNakerLink required that the network is configured to send multicast packets to and from the device. To do this, you need to create a model for your device, which extends from the python `AbstractVirtualVertex` class. This class requires that you pass in the `spinnaker_link_id` of the link to which your device is connected. Once you have

done this, you can create a PyNN population with the model. To use the device as input to a population, you can then simply use the device as a target in a projection. To output spikes to the device however, you cannot use a projection, as the device does not have a synaptic matrix. Instead, a function `activate_live_output_to(population, device)` is used; this create a link from the population to the device. Note that the spikes will not be filtered - every spike produced by the population will be sent to the device. These will be sent as 32-bit multicast packets.

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/workshop_material/external_devices_tasks/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/workshop_material/external_devices_tasks/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. It will respond to neuron ids. Send some spikes from the spike source array and look at the output from the script. The device doesn't care about the bits in the rest of the key - only the bottom 2 bits will be examined.

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/workshop\_material/external\_devices\_tasks/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_restricter.
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. Additionally, use `activate_live_output_to` to create a link from the population back to the device. 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 which will receive spikes from your device and those which will send spikes to your device.