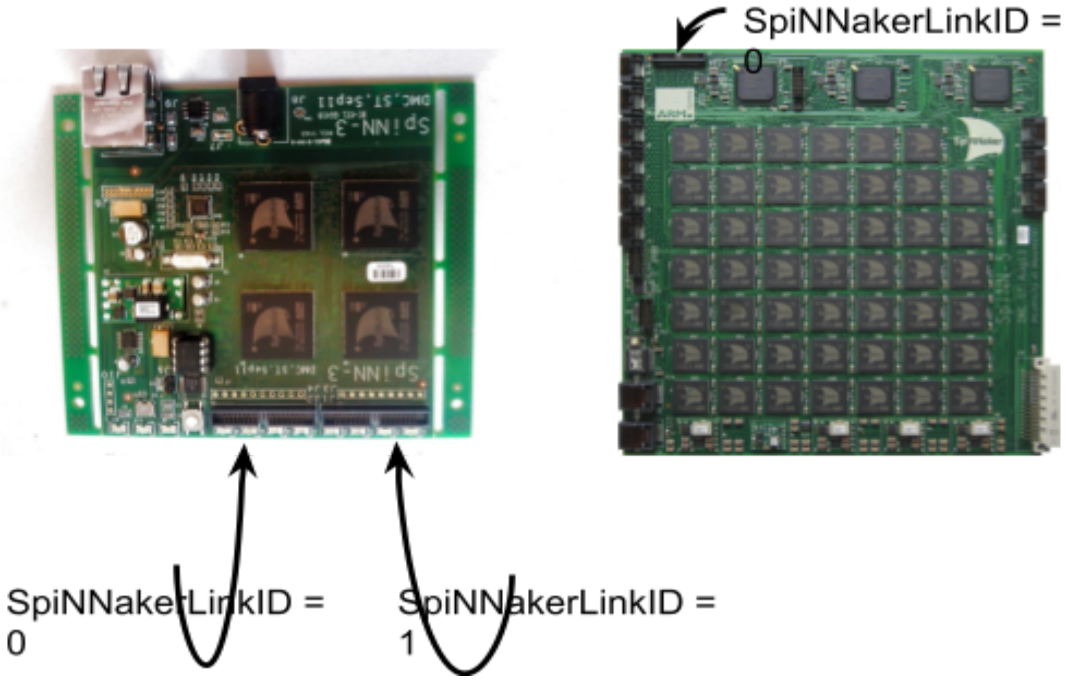
External Devices on SpiNNaker - Lab Manual

Introduction

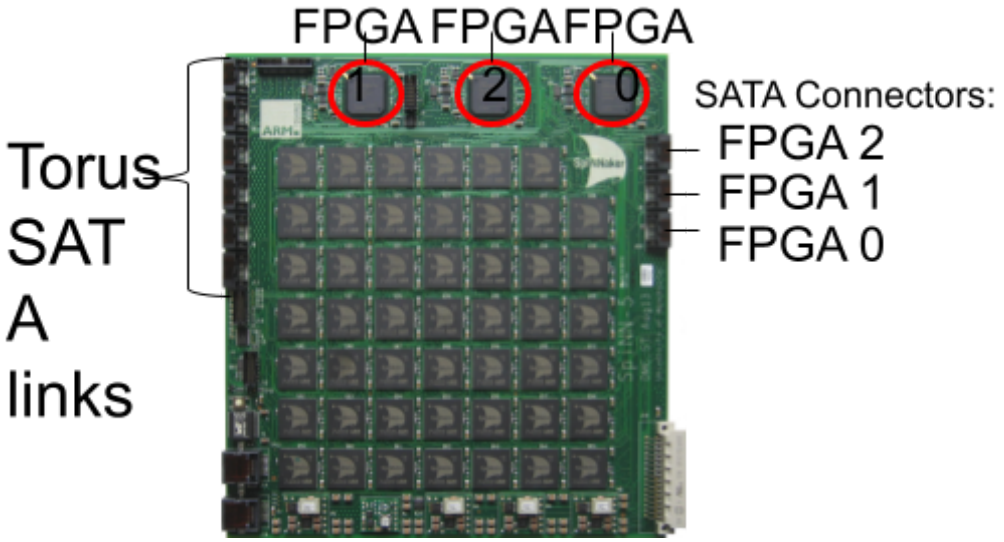
This manual will discuss the connection of external devices to SpiNNaker and how to tell the software that they are in use.

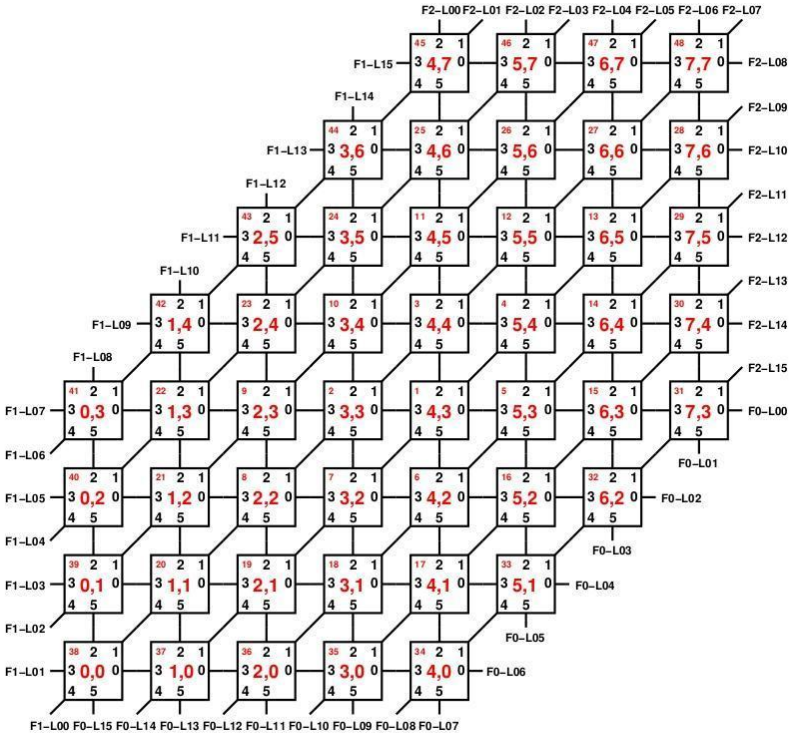
Physical Connection

The SpiNNaker boards have a number of places to which external devices can be connected. The first of these is the SpiNNakerLink connections. These are shown below.



Additionally, the 48-node boards have FPGAs which are linked to the SATA connectors. These are shown below.





The FPGAs are connected internally to a subset of the border chips on the board as shown above; the links at the edges of the board are coded as F<fpga-id>-L<fpga-link-id>, so F1-L04 is FPGA 1, FPGA link id 4.

The FPGA id and FPGA link ids, or the SpiNNakerLink ids are used to tell the software where a device has been connected. When a SpiNNakerLink is specified, the device is connected directly to this link. When an FPGA id is used, this indicates the FPGA to which the device is connected and which link of the FPGA it is connected to. The tools do not currently do any reprogramming of the FPGAs themselves, so the FPGA must have been configured in advance to forward packets between this link of the FPGA and the SATA link to which the device is connected.

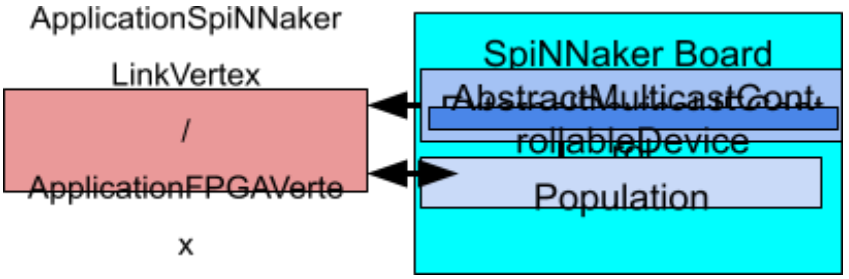
When a multi-board system is in use, it is important that you do not connect your device to the SpiNNakerLink unless you know that the FPGA has been disabled for that link. If you don't do this, you could damage the board. Devices can be connected to the FPGA SATA connectors as you wish, but without reconfiguring the FPGA, it is unlikely that any communication will occur.

When using a multi-board system, you must also be able to tell the software which board you are connecting your device to. This is done using the IP address of the ethernet-connected chip on the board. The board which contains chip 0, 0 will generally have the same IP address as the one you use to contact the board; if you don't specify a board address, it will be assumed that you want to use this board in any case, so unless you have multiple devices to connect, you may want to choose this board first.

Ethernet Connection

If you have a device that connects over Ethernet or some other means using the host machine, the software can also be configured to act as an intermediary between the device and SpiNNaker. No special connections are required in this case.

Software Connection for SpiNNaker Link and FPGA Devices



Device Specification

Once you have physically connected the device to the machine, the software needs to be told that that you want to add the device to a neural network. This is generally done by extending one of the models provided for external devices. These are:

- **pacman.model.graphs.application.ApplicationFPGAVertex** - this is used when the device is connected to an FPGA. You need to provide the **fpga_id**, the **fpga_link_id** and optionally the **board_address** when you are using a multi-board system, and the board connected to is not the first board.
- **pacman.model.graphs.application.ApplicationSpiNNakerLinkVertex** - this is used when the device is connected to a SpiNNaker Link. You need to provide the **spinnaker_link_id** and optionally the **board_address** when you are using a multi-board system, and the board connected to is not the first board.

As an example, you might have the following to represent a device connected to an FPGA:

```
import pyNN.spiNNaker as sim
from pacman.model.graphs.application import ApplicationFPGAVertex

class MyDevice(ApplicationFPGAVertex):
    def __init__(
            n_neurons, fpga_id, fpga_link_id, board_address=None, label=None):
        super(MyDevice, self).__init__(
            n_neurons, fpga_id, fpga_link_id, board_address, label)
```

Input

If you are using the device as an input device input a PyNN network, you will need to specify the multicast keys that the device will generate. This is done by creating a Python **class** for the device which extends one of the aforementioned interfaces in addition to the interface:

spinn_front_end_common.abstract_models.AbstractProvidesOutgoingPartitionConstraints

This requires the addition of a method called:

def get_outgoing_partition_constraints(self, partition)

This takes a parameter called **partition**; this can be ignored for the purposes of integration in PyNN. This function should return a **constraint** on the keys that are to be sent from this vertex. In general, you can return a fixed key and mask pair from this method indicating the base key and range of keys to use e.g.:

return [FixedKeyAndMaskConstraint([BaseKeyAndMask(0x12340000, 0xFFFF0000)])]

The example above would indicate that the device will send keys where the first 16 bits have the value 0x1234 and the rest of the bits can be set to any value between 0 and 0xFFFF (thus the range of keys the device is expected to send is 0x12340000 - 0x1234FFFF). It is not critical that the device uses all these keys, but it is essential that every key that the device will send is covered, otherwise packets sent by the device can interfere with the operation of the SpiNNaker system.

You can now use the device **as a source** in a PyNN network by providing it as a model in a Population. The device will then be treated as a source of spikes, like a SpikeSourceArray or SpikeSourcePoisson. Thus you can create a projection from the device to other Population objects in the network, where the device population is the source population.

Output

If you would like to send spikes to the device, this is done differently from other objects in the network. This is because a Projection expects to accept a connection describing the connectivity between the source and target neurons. Populations of neurons use the connection information to create a synaptic matrix; on reception of a spike this is used to work out which neurons are targeted by the spike. An external device is not assumed to have a synaptic matrix, and there is no standard way for the software to communicate the synaptic matrix to the device. Thus the device is expected to accept all the spikes from all the neurons, and process them accordingly. For this reason, a separate method is used to send spikes from a PyNN Population object to a device. This is:

sim.external_devices.activate_live_output_to(<source_population>, <destination_device>)

where **<source_population>** is a standard PyNN Population object (which can include other devices and input sources), and **<destination_device>** is the device population, a population created with the device as the model.

The keys that the device will receive in this case are those that have been assigned to the source population. If the device has a specific set of keys that need to be sent to control it, this will be described in more detail below.

Commands

Commands are SpiNNaker multicast messages with or without payloads that can be sent to a device at the start and end of a simulation, or at specified times during a simulation. This can be used to set up and cleanly stop an external device. In these contexts, you can tell a device the multicast keys to use in the simulation, and you can tell it to start transmitting at the start of the simulation, and stop transmitting at the end. This can be important when operating SpiNNaker, since if your device is sending data packets into SpiNNaker, it might not be able to perform normal functions, such as booting the system.

If you have a device that supports commands, you can extend your device class with:

spinn_front_end_common.abstract_models.AbstractSendMeMulticastCommandsVertex

This requires the following properties to be implemented, each of which returns a list of:

spinn_front_end_common.utiltiy_models.MultiCastCommand(
key, payload=None, time=None)

- **start_resume_commands** - returns a list of MultiCastCommand instances to send at the start of the simulation, or when the simulation resumes after a pause. The time field is ignored.
- **pause_stop_commands** - returns a list of MultiCastCommand instances to send at the end of simulation, or when the simulation pauses. The time field is ignored.

- **timed_commands** - return a list of MultiCastCommand instances to send at specified times during the simulation. The time field must be specified.

Any of these can return an empty list of commands (most commonly the timed_commands does this).

Device Control During Simulation

In addition to the above support for handling direct output of spikes to devices, the software can also be configured to handle the sending of commands to a device based on the state of the simulation, in terms of the membrane voltage of a LIF neuron. These commands will consist of multicast messages with a payload, where the payload is the membrane voltage. The messages can be configured to be sent at multiples of the timestep of the simulation to reduce the amount of data being sent over the SpiNNaker network.

The neuron model to use to control devices is:

sim.external_devices.ExternalDeviceLifControl

This has the parameters of the IF_curr_exp neuron, with the exception of v_thresh (since it has no threshold voltage and never spikes). Additionally, the following parameters must be specified:

- **devices** - the list of devices that will be controlled by this population. The length of devices must be the same as the number of neurons in the population. These are detailed below.
- **create_edges** - indicates whether edges to the devices should be added to the network. This is for compatibility with Ethernet devices (see later). Set to True for SpiNNakerLink and FPGA devices.

Each device specified in the parameter **devices**, in addition to extending either the ApplicationSpiNNakerLinkVertex or ApplicationFPGAVertex, should additionally extend:

spynnaker.pyNN.external_device_models.AbstractMulticastControllableDevice

This requires the class to have the following properties:

- **device_control_partition_id** - the id to give the partition of edges going to this device. This can be anything such as the name of the device.
- **device_control_key** - the key to use in the multicast packet to be sent to the device.
- **device_control_uses_payload** - True if the device will take the membrane voltage as a payload, False if the key will be sent by itself. It is expected that most devices will want to set this to true to receive the membrane voltage in the payload.
- **device_control_min_value** - the minimum value of membrane voltage that the device accepts. If the voltage is below this value, this value is sent instead.
- **device_control_max_value** - the maximum value of membrane voltage that the device accepts. If the voltage is above this value, this value is sent instead.
- **device_control_timesteps_between_sending** - the number of timesteps between the sending of the packets, to reduce the bandwidth used.

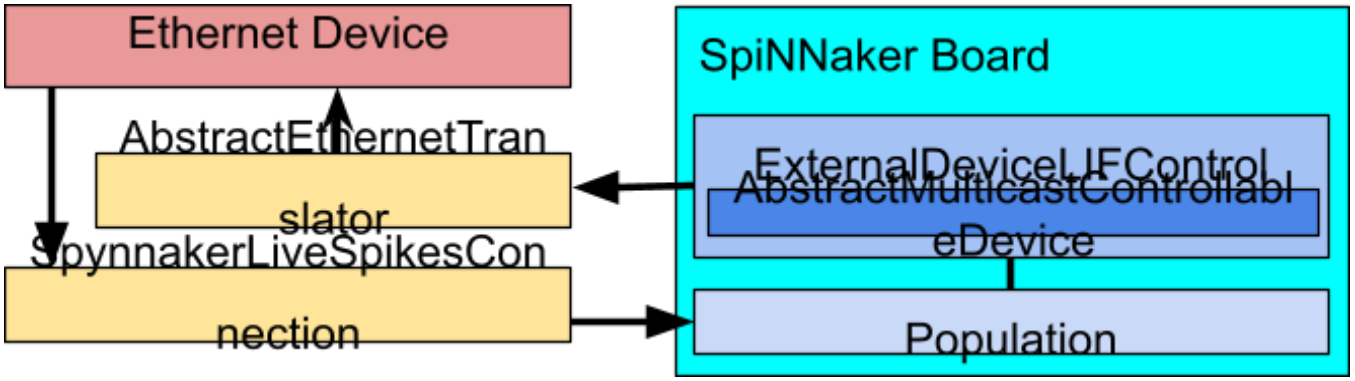
This LIF-based model can now be used as a Population in a simulation, and as a target population of a standard PyNN Projection. Spikes received by the population will increase the membrane voltage of neurons in the population if they are connected via an excitatory connection, or decrease the membrane voltage if connected via an inhibitory connection, as normal. It is important to realise though that a Population object created with this model will never spike. It is not an error to use it as the pre-population in a Projection, but no spikes will be sent across the projection in that case. Note that this population can now be recorded like any other population, unlike the devices themselves.

The format of the membrane voltage sent as the payload is S1615 format (this is a 32-bit number with the first bit being the sign, the next 16-bits are the integer part of the number and the remaining 15-bits are the fractional part of the number). The default parameters therefore set the rest and reset voltages of the neuron to 0 (e.g. when using a motor where the membrane voltage is the speed of the motor, this would mean the speed would be 0 without any input).

Warning about Keys

There are two potential sources of keys that can be received by the devices: those sent by commands at the start and end of simulation, and those sent to the device during simulation containing the membrane voltage. These two sources of keys must generate distinct keys e.g. if you are using a command at the start that sets a motor speed to 0, and then you want to set the motor speed using the voltage during simulation, your device needs to accept different keys for this. This is often done by having your device ignore part of the key e.g. it might use the bottom 16-bits to identify the command it is being asked to perform, but ignore the top 16-bits.

Software Connection for Ethernet and Other Devices



Output

If your device only works over an Ethernet connection, or some other connection via the host machine, this can be controlled using the Live I/O discussed in this lab. The software does, however, have some support for using these devices with the aforementioned LIF neuron membrane voltage output. To make this work, the device is defined as a class which, as described for SpiNNakerLink and FPGA devices, extends the interface:

spynnaker.pyNN.external_device_models.AbstractMulticastControllableDevice

See above for a description of the properties that the implementation must provide. Note that the device does **not** now have to extend ApplicationSpiNNakerLinkVertex or ApplicationFPGAVertex. However the device still has to produce keys for the commands. These can be arbitrary values, so long as they don't clash with other keys in use.

Once the device is defined, you then need to define a class that extends:

spinnaker.pyNN.external_device_models.AbstractEthernetTranslator

This is the class that will communicate with your device, and so should contain all the mechanisms through which you do this. The class requires the implementation of a single method:

def translate_control_packet(self, multicast_packet)

This method takes a multicast packet, which will have a properties of **key** and **payload** indicating the key and payload of the packet received. The key will be the key returned by the device_control_key that you defined as part of the definition of your device. The payload will contain the membrane voltage value.

Note that you can also make your device extend the class:

spinn_front_end_common.abstract_models.AbstractSendMeMulticastCommandsVertex

This will allow the device to be sent commands at the start and end of simulation. These commands will also be sent via the above translator instance, so the translator must also recognise these packets. As with the other devices, the keys used for start and stop commands must be different from those used to control the device, though with an Ethernet device, these keys are arbitrary in any case.

Once these parts have been defined, an instance of **sim.external_devices.ExternalDeviceLifControl** is again created, with parameters as previously described, but with **create_edges** set to False, and an additional parameter **translator**, which is an instance of the translator created above.

The final instantiation of the device is via **sim.external_devices.EthernetControlPopulation**. This is used **in place of** sim.Population, **not** as a model. This method takes the following parameters:

- **n_neurons** - the number of neurons in the population. This should be the same as the number of devices passed to the ExternalDeviceLifControl model.
- **model** - this is the ExternalDeviceLifControl instance defined above.

The result of this call can be used as the target population of a PyNN Projection to send spikes to the device. It is valid to use it as a source population but the Projection will then pass no spikes.

Input

For Ethernet Input, again Live I/O can be used as described elsewhere. The software also provides additional interfaces to link the device more closely to the software and make it easier to use as a single unit.

To take advantage of this functionality, the Ethernet input device should be defined in a class which extends:

spynnaker.pyNN.external_device_models.AbstractEthernetSensor

This class requires that the following methods are implemented:

- **get_n_neurons** - return the number of neurons the sensor provides.
- **get_injector_parameters** - return any parameters to pass to the SpikeInjector.
- **get_injector_label** - return the label to give to the SpikeInjector.
- **get_translator** - return an AbstractEthernetTranslator as defined for output. This is used for start and stop commands, so you can return None if the class doesn't also extend AbstractSendMeMulticastCommandsVertex.
- **get_database_connection** - return a SpynnakerLiveSpikesConnection that is to be used to send the spikes into SpiNNaker.

If desired, you can write a class which extends SpynnakerLiveSpikesConnection and gathers data from your device and uses the send_spikes method to inject spikes into the network based on the data. This can then be returned from get_database_connection. Alternatively, you can create a SpynnakerLiveSpikesConnection instance elsewhere and return this directly.

Once this has been implemented, you can instantiate the device using:

sim.external_devices.EthernetSensorPopulation

This is again used in place of a pyNN Population object and **not** as a model. This requires a single paramerter **device** which is the device you have defined above.

The result of this call be be used as the source population of a projection. It is not valid to use this as a target population.

Pushbot

The TUM Pushbot is a commonly used robotics platform which is compatible with SpiNNaker over Ethernet and SpiNNakerLink with an additional SpiNNaker adapter. The Pushbot therefore is a good example of the use of the various interfaces described above.

The Pushbot has several output devices, including two motors, a laser, two LEDs, and a speaker. These are each considered to be different devices from the point-of-view of the tools. To help with key allocation, the pushbot implementation has a MunichIOProtocol instance, which gives the keys for various commands given a base key.

The Pushbot has several sensors, but so far only the retina has been implemented in the tools. When used over SpiNNakerLink, the SpiNNaker board protocol will downsample the retina into various formats. The Ethernet connection to the Pushbot doesn't support this, so the software is designed to do this on the host machine between receiving spikes from the retina and sending them on to SpiNNaker.

The tools also include a Pushbot retina visualiser. This can receive spikes from the Pushbot retina and display them in a graphical window, which shows how the retina works based on the changes in light.

Tasks

These tasks will take you through the process of interacting with external devices. You don't actually have to have a device connected to work through this process, or to run some scripts. You may see some warning messages because of this, but the scripts should still otherwise run correctly.

Task 1.1 [Easy]

Create a network which sends spikes from a Poisson spike source to a device which extends ApplicationSpiNNakerLinkVertex with a single neuron on spinnaker link 0. Note that when you run the network, you should get a message like the following:

```
2017-09-22 20:28:28 WARNING: The reinjector on 0, 0 has detected that 76242 packets were dumped from a outgoing link of this chip's router. This often occurs when external devices are used in the script but not connected to the communication fabric correctly. These packets may have been reinjected multiple times and so this number may be a overestimate.
```

The reason for this is that there isn't actually a device connected to your board. However, this message indicates that the packets are correctly routed towards the spinnaker link.

Task 1.2 [Easy]

Try changing the spinnaker link to 1. If you have a 4-node board, this should result in a similar message but referencing chip 1, 0. If you are using a 48-node board, you should get an error since there isn't a second spinnaker link on the board.

Task 2.1 [Medium]

Create another device class that extends ApplicationSpiNNakerLinkVertex and AbstractProvidesOutgoingPartitionConstraints. Set up the device up so that it sends with a base key of 0x12340000 and mask 0xFFFF0000. Set up a network with a projection from the device to an IF_curr_exp population. You will not get any spikes; to verify that the network is working, look inside the folder "reports" for the folder with the latest date. Inside this folder, look at the file:

"run1/virtual_key_space_information_report.rpt"

This should show you the key and mask assigned to your device.

Task 2.2 [Medium]

Extend the class further to implement AbstractSendMeMulticastCommandsVertex. Add some start and end commands. Again look inside the report and see that the commands have been added.

Task 3 [Hard]

Create another device class, this time extending ApplicationSpiNNakerLinkVertex and AbstractMulticastControllableDevice. Create a script with a Population that uses a ExternalDeviceLifControl as a model, which takes the device you just created. Record the membrane voltage from the device. Connect a Poisson spike source to the device, run for some number of milliseconds and then print or graph the membrane voltage.

Task 4 [Hard]

Create another device class which extends AbstractMulticastControllableDevice but nothing else. Create a translator class that extends AbstractEthernetTranslator, and in the implementation prints out the key and payload of the multicast message. Add the device to an ExternalDeviceLifControl instance along with the translator. Create an EthernetControlPopulation, and feed this with a Poisson spike source. Record the membrane voltage of the control population, run the simulation for some number of milliseconds and then print or graph the membrane voltage.

Task 5 [Very Hard]

Create a class which extends AbstractEthernetSensor and has a single neuron. This class should create its own SpynnakerLiveSpikes connection, and register a method against this to send a spike to neuron 0 at start up (probably after a short pause e.g. 0.01 seconds to make sure the simulation is actually running). Be careful that the label given to the connection is the same as that returned from the sensor get_injector_label() method, and make sure this same connection is returned from the get_database_connection() method. The sensor doesn't send start or stop commands, so it doesn't need a translator (i.e. this method can return None), and no additional injector parameters are required. Connect the sensor to a standard LIF population with a single neuron. Record the voltage from this population and make sure that it changes in response to the single injected spike.

Note you may need to add a new spynnaker.cfg file to the folder where you are running the script, with the following contents:

```
[Database]
create_database = True
```

This is to overcome a bug in the detection of when the database needs to be created.