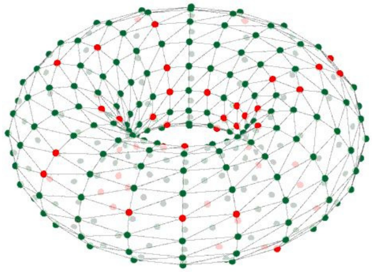
Introduction to the Graph Front End Functionality



Alan Stokes, Andrew Rowley

SpiNNaker Workshop
September 2016

European Research Council
Established by the European Commission

Human Brain Project

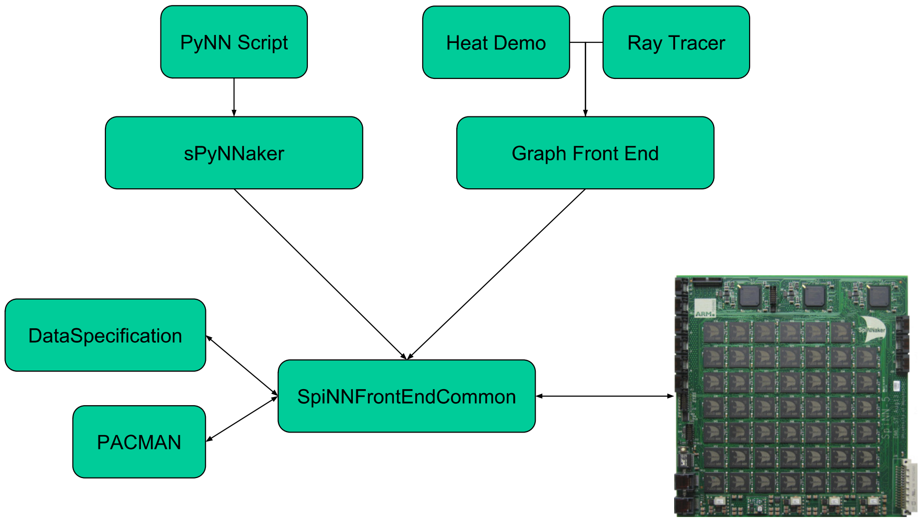
EPSRC

SpiNNaker

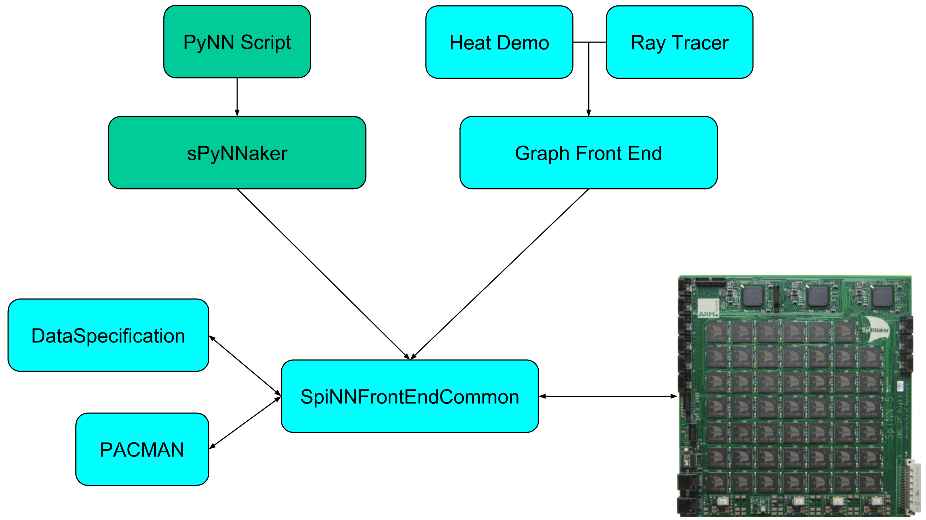
Contents

- The Graph Front End (GFE) interface
- Simple Usage of the GFE
- The Graph in the GFE
- Data Generation
- Binary Specification
- Writing and building simple C code

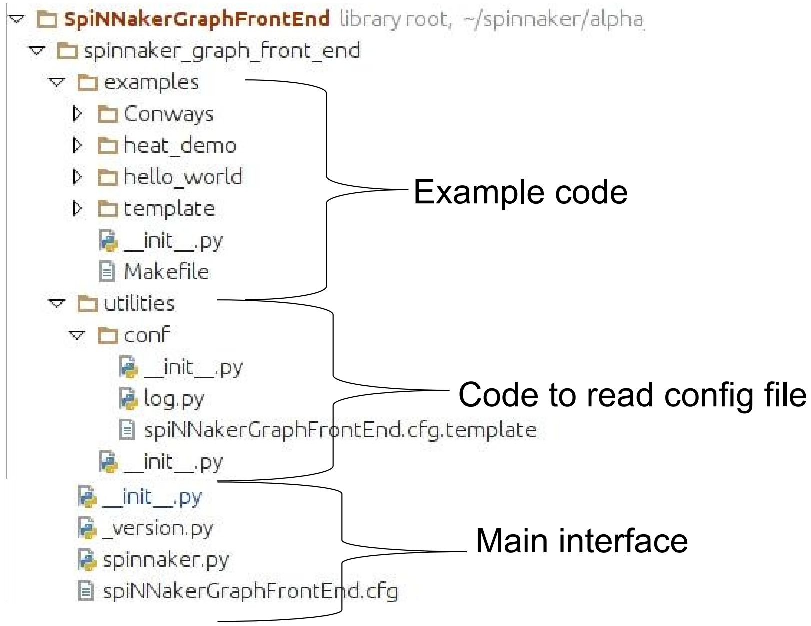
Software modules



Software modules covered here



GFE structure



Skeleton Functionality

```python
import spinnaker_graph_front_end as front_end

# set up the front end
front_end.setup()

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```

ConfigurationException:
There needs to be a graph which contains at least one vertex for the tool chain to map anything.

Main interface functions

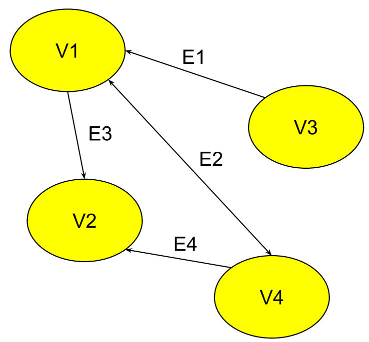
import spinnaker_graph_front_end **as** p

p.setup() Sets up the software stack so that it has read the configuration file and created whatever data objects are required.

p.run(duration) Runs the simulation for a given time period (microseconds).

p.stop() Closes down the application that is running on the SpiNNaker machine and does any housekeeping needed to allow the next application to run correctly.

PACMAN Graph



Has a 1:1 ratio between vertices and SpiNNaker core.

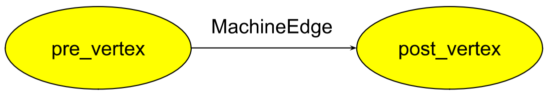
Basic script to add machine vertices into the graph

```
import spinnaker_graph_front_end as front_end

from spinnaker_graph_front_end.examples.Conways.conways_cell import \
  ConwayMachineCell

# set up the front end
front_end.setup()

for count in range(0, 800):
   front_end.add_machine_vertex_instance(
      ConwayMachineCell(label="cell{}".format(count)))

# run the simulation for 5 seconds
front_end.run(5000)

# clean up the machine for the next application
front_end.stop()
```


Creating a new type of machine vertex

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer

class ConwayMachineCell(MachineVertex):
  """ Cell which represents a cell within the 2D grid
  """

  def __init__(self, label):

     # construct the resources this cell uses and instantiate superclass
     resources = ResourceContainer()
     MachineVertex.__init__(self, resources, label)
```


Adding edges to the machine graph



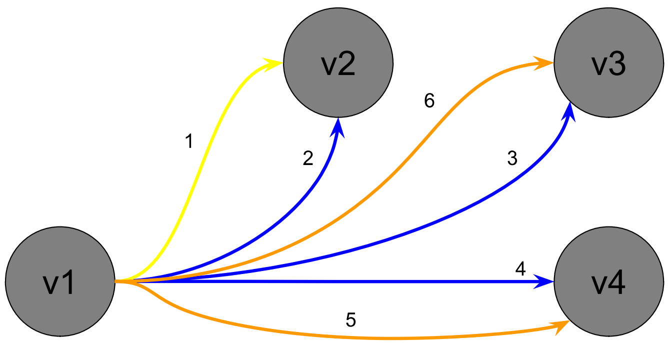
1. The main edge type available is MachineEdge.
2. This can be extended to add application specific data into.
3. Most important inputs are:
   i. **pre_vertex**: The source of the edge.
   ii. **post_vertex**: The destination of the edge.
4. Every edge in a graph is associated with a **partition_id**.

Basic Script adding edges

```
import spinnaker_graph_front_end as front_end
.......

# build and add vertices to the graph
vertices = list()
for count in range(0, 100):
  vertices.append(ConwayMachineCell("cell{}".format(count)))
  front_end.add_machine_vertex_instance(vertices[count])

# build an edge between two vertices
front_end.add_machine_edge_instance(
   MachineEdge(verts[0], verts[1]), "State")
                                    Partition id
front_end.run(5000)

front_end.stop()
```

MANCHESTER 1824

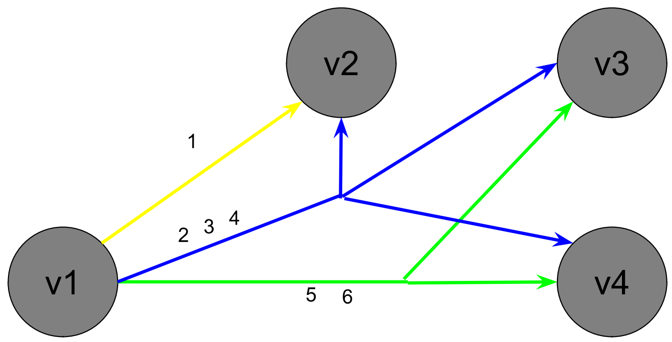
Adding edges to the application graph: Partitions



Edge 1 resides in partition A
Edges 2,3 and 4 reside in partition B
Edges 5 and 6 reside in partition C

MANCHESTER 1824

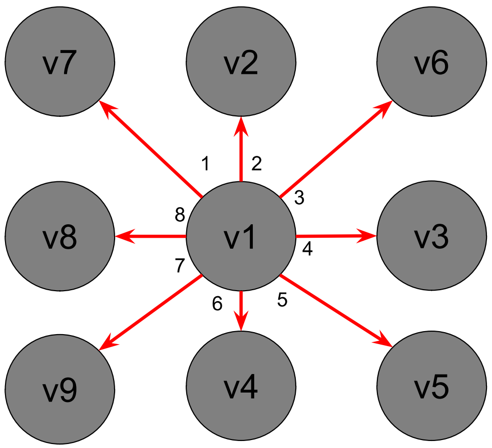
Adding edges to the application graph: Partitions



Edge 1 transmits information about **hotdogs**.
Edges 2,3 and 4 transmits information about **cats**.
Edges 5 and 6 transmits information about **bacon**.

MANCHESTER 1824

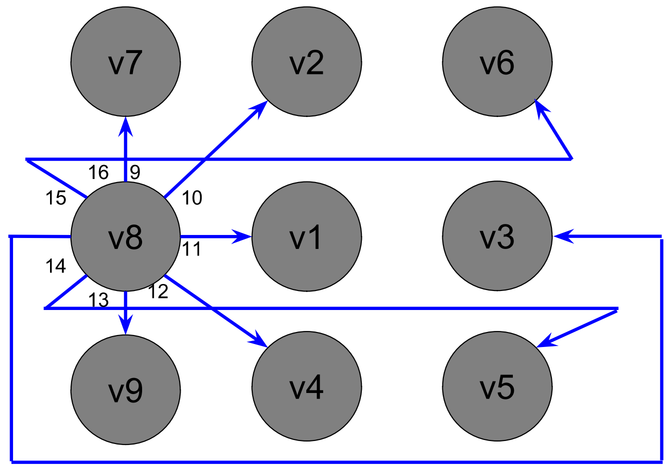
Conways: partitions.



Edges 1,2,3,4,5,6,7,8 transmits v1’s **state** data.

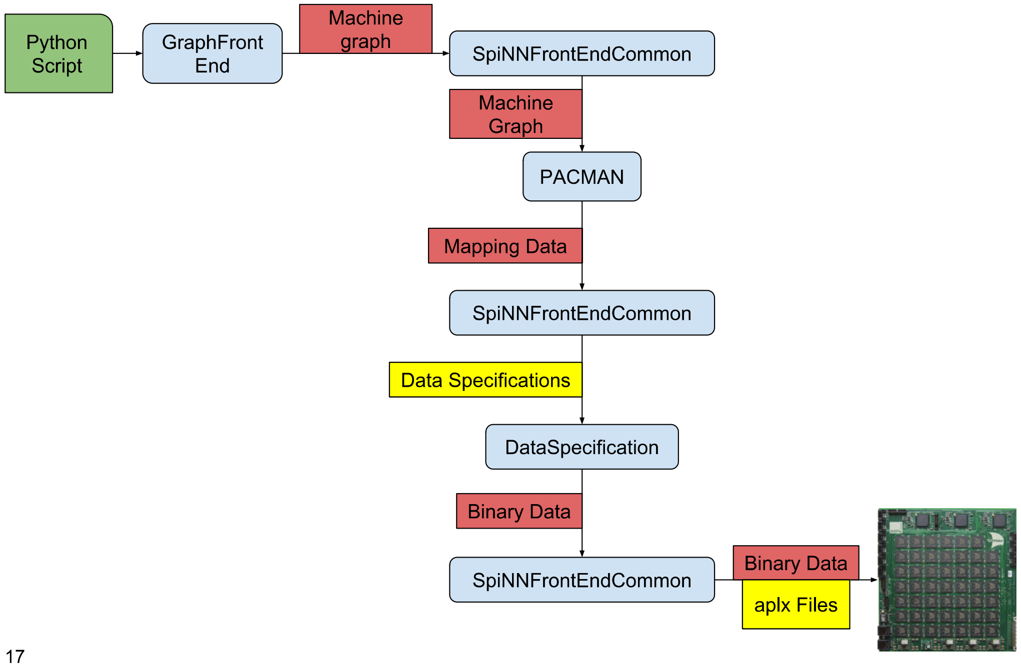
MANCHESTER 1824

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmits v8’s **state** data.

Workflow of the GFE



Data generation

1. Converts application data within a vertex into data stored on the SpiNNaker machine via SDRAM.
2. Supports separating the SDRAM into data regions
3. Supports writing data as scalars, arrays etc.
4. Common commands:
    i. reserve_memory_region()
    ii. switch_write_focus()
    iii. write_value()
    iv. write_array()
    v. comment()
    vi. close_spec()

Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
   region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
   region=1, size=8, label="inputs")
```


Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
   region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
   region=1, size=8, label="inputs")

# add simulation.c interface data
spec.switch_write_focus(0)
spec.write_array(simulation_utilities.get_simulation_header_array(
   self.get_binary_file_name(), machine_time_step, time_scale_factor))
```

Needed for simulation.c

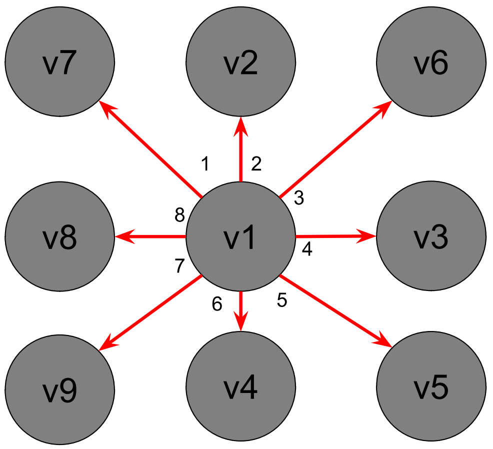
Data generation

```
...
def generate_machine_data_specification(
        self,  spec, placement, machine_graph, routing_info, Iptags,  reverse_iptags,
        machine_time_step, time_scale_factor):

# Reserve SDRAM space for memory areas:
spec.reserve_memory_region(
   region=0, size=constants.SYSTEM_BYTES_REQUIREMENT, label='system')
spec.reserve_memory_region(
   region=1, size=8, label="inputs")

# add simulation.c interface data
spec.switch_write_focus(0)
spec.write_array(simulation_utilities.get_simulation_header_array(
   self.get_binary_file_name(), machine_time_step, time_scale_factor))

# application specific data items
spec.switch_write_focus(region=1)
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)
```


Data generation

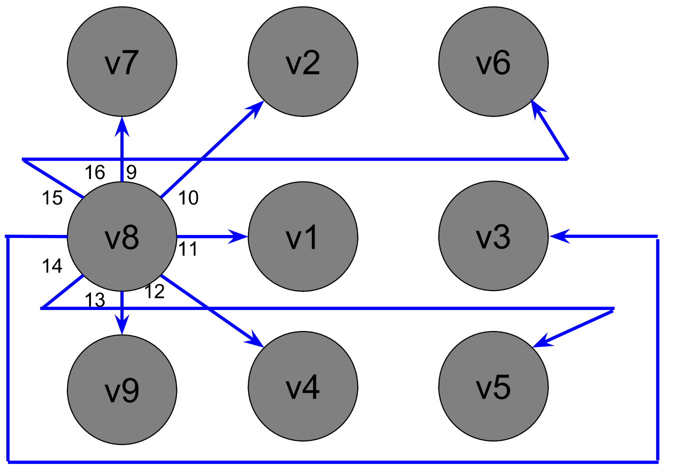
```
...
spec.comment("writing initial state for this conway element \n")
spec.write_value(data=self._state)

# write the routing key needed for my transmission
spec.comment("writing the routing key needed to transmit my state \n")
spec.write_value(data=routing_info.get_first_key_from_pre_vertex(self, "State"))

# close the spec
spec.comment("closing the spec \n")
spec.close_spec()
```


Conways: partitions.



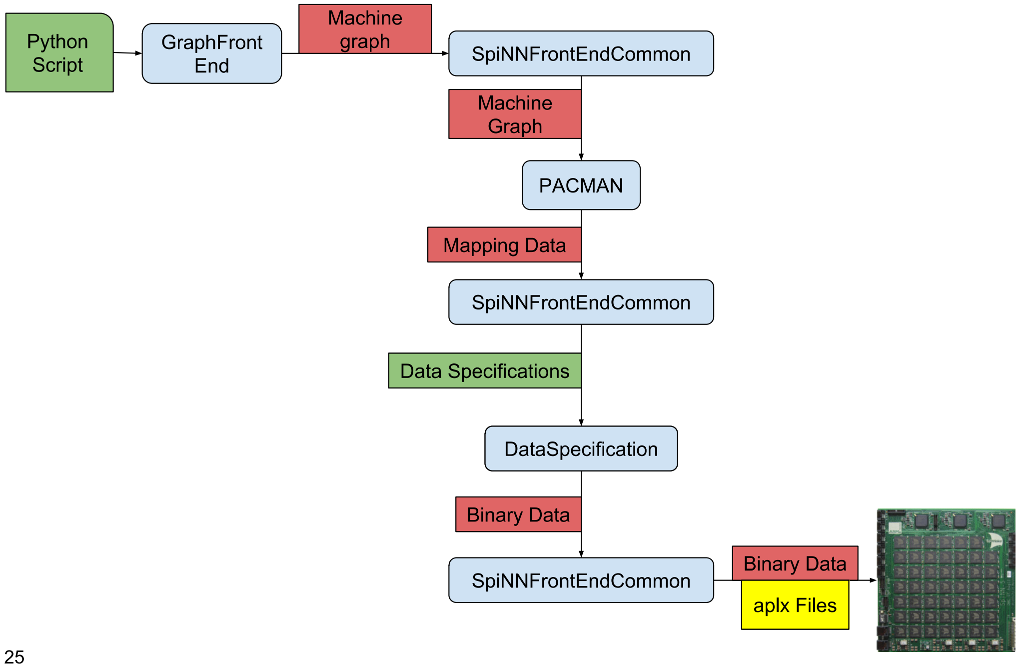
Edges 1,2,3,4,5,6,7,8 transmits with routing key 0.

Conways: partitions.



Edges 9,10,11,12,13,14,15,16 transmit with routing key 1.

Workflow of the GFE



Binary Executables

AbstractHasAssociatedBinary

```
def get_binary_file_name(self)
```

AbstractStartsSynchronized

- After loading binary, CPU state will be in SYNC0

AbstractBinaryUsesSimulationRun(AbstractStartsSynchronized)

- The binary uses the simulation environment provided by the tools

Binary Executables

```
from pacman.model.graphs.machine.impl.machine_vertex import MachineVertex
from pacman.model.resources.resource_container import ResourceContainer
from spinn_front_end_common.abstract_models.abstract_has_associated_binary\
    import AbstractHasAssociatedBinary
from spinn_front_end_common.abstract_models.abstract_binary_uses_simulation_run\
    import AbstractBinaryUsesSimulationRun

class ConwayMachineCell(
        MachineVertex, AbstractHasAssociatedBinary,
        AbstractBinaryUsesSimulationRun):
    """ Cell which represents a cell within the 2D grid
    """

    def __init__(self, label):

        # construct the resources this cell uses and instantiate superclass
        resources = ResourceContainer()
        MachineVertex.__init__(self, resources, label)

    def get_binary_file_name(self):

        # return the binary name of this vertex
        return "conways.aplx"
```


Linking Aplx files to Python

1. Compiled version of c code.
2. This code runs on SpiNNaker.
3. Is linked to your python classes through get_binary_file_name(self) of the vertex
4. How to write event driven C code for SpiNNaker is discussed in **Event Driven Simulations**.
5. We will cover the interfaces provided by the SpiNNFrontEndCommon (SFEC) module for the c code.

Example c code

```c
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

    // get address of simulation data
    address_t address = data_specification_get_data_address();
```

 # gets the address where all the data for this core is stored.

address_t data_specification_get_data_address();

Example c code

```c
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {

    // get address of simulation data
    address_t address = data_specification_get_data_address();

    // get the address of the system region
    address_t system_region = data_specification_get_region(0, address);
```

 # gets the address of the start of a given data region

address_t data_specification_get_region(uint32_t region, address_t data_address)

Example c code

```c
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
    ...
    if (!simulation_initialise(
            system_region, APPLICATION_NAME_HASH,
            &timer_period, &simulation_ticks,
            &infinite_run, SDP_PRIORITY,
            NULL, NULL)) {
        log_error("Error in initialisation - exiting!");
        rt_error(RTE_SWERR);
    }
```

 # sets up the system to interact with the SFEC simulation control functionality.

```c
bool simulation_initialise(
        address_t address, uint32_t expected_application_hash,
        uint32_t* timer_period, uint32_t *simulation_ticks_pointer,
        uint32_t *infinite_run_pointer, int sdp_packet_callback_priority,
        prov_callback_t provenance_function, address_t provenance_data_address)
```


Example c code

```c
static uint32_t timer_period, simulation_ticks, infinite_run = 0;
static uint32_t time;
static uint32_t SDP_PRIORITY = 1, TIMER_PRIORITY = 2;

void c_main(void) {
    ...

    // Set timer_callback period
    spin1_set_timer_tick(timer_period);

    // Set timer_callback
    spin1_callback_on(TIMER_TICK, timer_callback, TIMER_PRIORITY);

    // Set time to UINT32 MAX to wrap around to 0 on the first timestep
    time = UINT32_MAX;

    simulation_run();
}
```

 # main entrance for the event driven nature of the SpiNNaker machine

void simulation_run()

Example c code

```c
// Callbacks
void timer_callback(uint unused0, uint unused1) {

    // check if the simulation has run to completion
    if ((infinite_run != TRUE) && ((time + 1) >= simulation_ticks)) {
        simulation_exit();
    }
    time++;
}
```

```
# Used once you have finished your simulation
void simulation_exit()
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/
```

Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c
```


Makefile

```
MAKEFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(dir $(MAKEFILE_PATH))

SOURCE_DIR := $(abspath $(CURRENT_DIR))
SOURCE_DIRS += $(SOURCE_DIR)

APP_OUTPUT_DIR := $(abspath $(CURRENT_DIR))/

BUILD_DIR = build/

APP = conways

SOURCES = conways.c

include $(SPINN_DIRS)/make/Makefile.SpiNNFrontEndCommon
```


Summary

1. How to use the GFE interface.
2. The machine graph supported by the GFE.
3. Adding vertices, edges and partitions to the machine graph.
4. Data Specification for the graph.
5. Binary Specification.
6. Building and making basic C code.