Rig Documentation

Release 2.4.1

Project Rig

May 13, 2018

Contents

Rig is a Python library which contains a collection of complementary tools for developing applications for the massively-parallel SpiNNaker architecture. First and foremost, Rig aims to present a light-weight, well tested and well documented interface for SpiNNaker application developers.

CHAPTER 1

Getting started

If you're new to Rig, here are two options for getting started: If you're feeling impatient and want to start playing, take a look at *some of the ten-line quick-start example programs*. Alternatively the *'hello world' to circuit simulator tutorial* gives a detailed introduction to building real-world SpiNNaker applications using Rig (still in under 400 lines of heavily commented Python).

1.1 Installation

Note: Since Rig is a library rather than a standalone tool, most end-users will find that it is automatically installed as a dependency of some other application which they have installed, rendering these steps unecessary.

1.1.1 From PyPI via `pip` (Recommended)

The latest stable release can be installed from the Python Package Index using:

```
$ pip install rig
```

Note that if you do not already have Numpy installed, this will be downloaded by the above command and may take some time to install.

1.1.2 From source

You can install Rig from downloaded source code using setuptools as usual:

```
$ git clone https://github.com/project-rig/rig.git rig
$ cd rig
$ python setup.py install
```

If you intend to work on Rig itself, take a look at the DEVELOP.md file in the repository for instructions on setting up a suitable development environment and running tests etc.

1.1.3 Optional Extras

The following extra packages may also be installed in addition to Rig to enable additional functionality.

`rig_c_sa` (for faster placement)

```
$ pip install rig_c_sa
```

The rig_c_sa library is used by the `CKernel` for the *simulated annealing placement algorithm*. This kernel, written in C, can be 50-150x faster than the `PythonKernel` supplied in the basic Rig installation.

1.2 Tutorial: From 'hello world' to a full application

In this tutorial we'll walk through the process of building a SpiNNaker application using the Rig library. This series of tutorials builds up from 'hello world' to eventually building full-blown digital circuit simulator. This tutorial is aimed at people who wish to build applications for SpiNNaker and gives a broad overview of how a SpiNNaker application might be structured and how Rig can be used to handle the boring details of interacting with and using a SpiNNaker machine.

Note: This tutorial presumes you are moderately familliar with Python and C and have some basic familiarity with the 'Spin1 API' and SARK libraries used to write software which runs on SpiNNaker itself.

1.2.1 00: Introduction

In this tutorial we'll walk through the process of building a SpiNNaker application using the Rig library. This series of tutorials builds up from 'hello world' to eventually building full-blown digital circuit simulator.

In *part 01* we build a simple hello world application demonstrating how Rig can be used to load and run programs on SpiNNaker.

In parts *02* and *03* we get the hang of reading and writing data into SpiNNaker.

In *part 04* we use what we have learned to build a proof-of-concept circuit simulator. This implementation can only simulate one circuit and uses hand-written routing tables and manually assigns work to SpiNNaker's processors.

In *part 05* we rewrite our circuit simulator as if it were a real application. We design a simple API for describing circuits and use Rig's automatic place and route facilities to automatically map our circuit simulations onto the SpiNNaker machine.

The digital circuit simulator we'll be building in this tutorial closely follows the program structure used in real-world SpiNNaker applications. Though only a very small program (under 400 lines of heavily annotated Python) our simulator goes through almost all of the steps real neural simulation applications do, including implementing a high-level domain-specific 'language' for describing simulations. With luck, after completing these tutorials you will have an understanding of how to go about building your own SpiNNaker applications.

Before getting stuck in, we'll take a look at how a typical SpiNNaker application is structured and highlight how Rig fits into this picture.

SpiNNaker applications and Rig

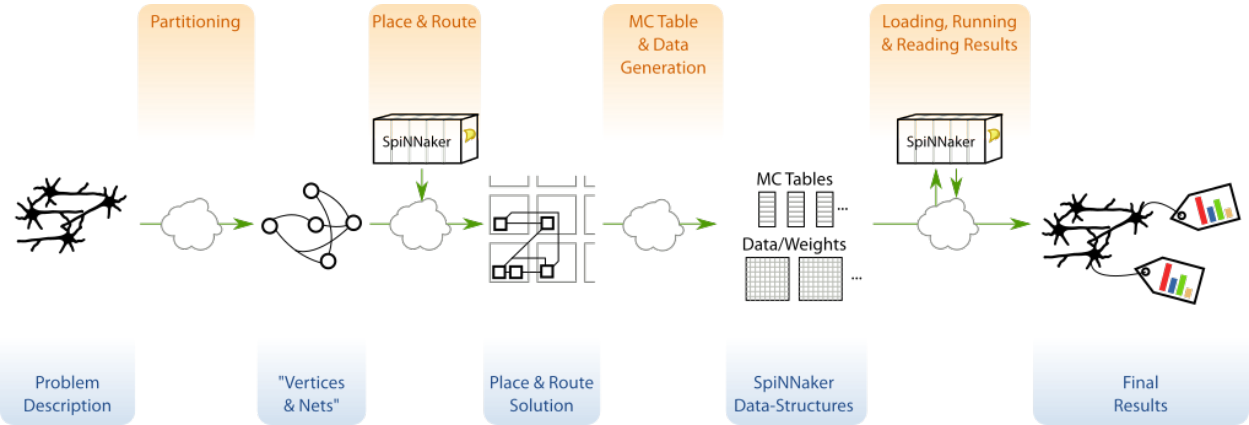
Though SpiNNaker is usually described as a 'computer', in practice most applications would be more accurately described as using SpiNNaker as a co-processor or an accelerator. As a result, most applications are written in two parts:

SpiNNaker application kernels A collection of small C (or C++) programs which run on SpiNNaker's application cores. These programs do the 'real' work in most applications, for example simulating neural behaviour, recording results, or playing back artificial stimuli.

Host application A program which runs on a host computer connected to your SpiNNaker machine, commonly written in Python. This program handles the conversion of high-level user input into raw data the SpiNNaker application kernels can process, loads software and data onto the SpiNNaker machine and retrieves and processes results.

Though the SpiNNaker application kernel is responsible for most of the actual computation, the host program is often more complex. Rig is a library which helps with the process of writing host applications by providing tools and functions which simplify the task of interacting with and programming a SpiNNaker machine.

In this tutorial we'll be building an application which follows the program structure used by the host programs of many existing neural simulation tools (e.g. Nengo SpiNNaker and PyNN SpiNNaker). This program structure is illustrated in the figure below:

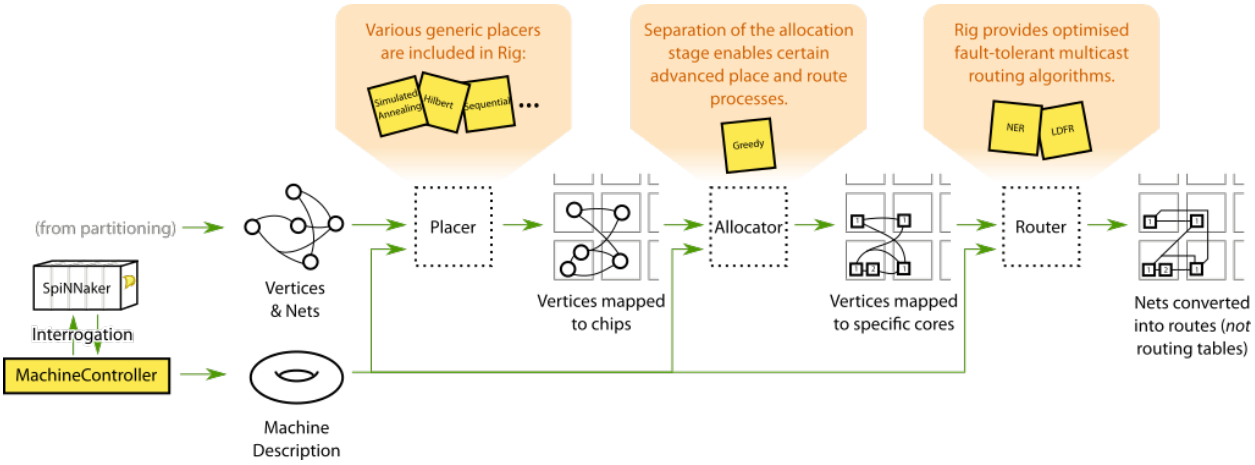


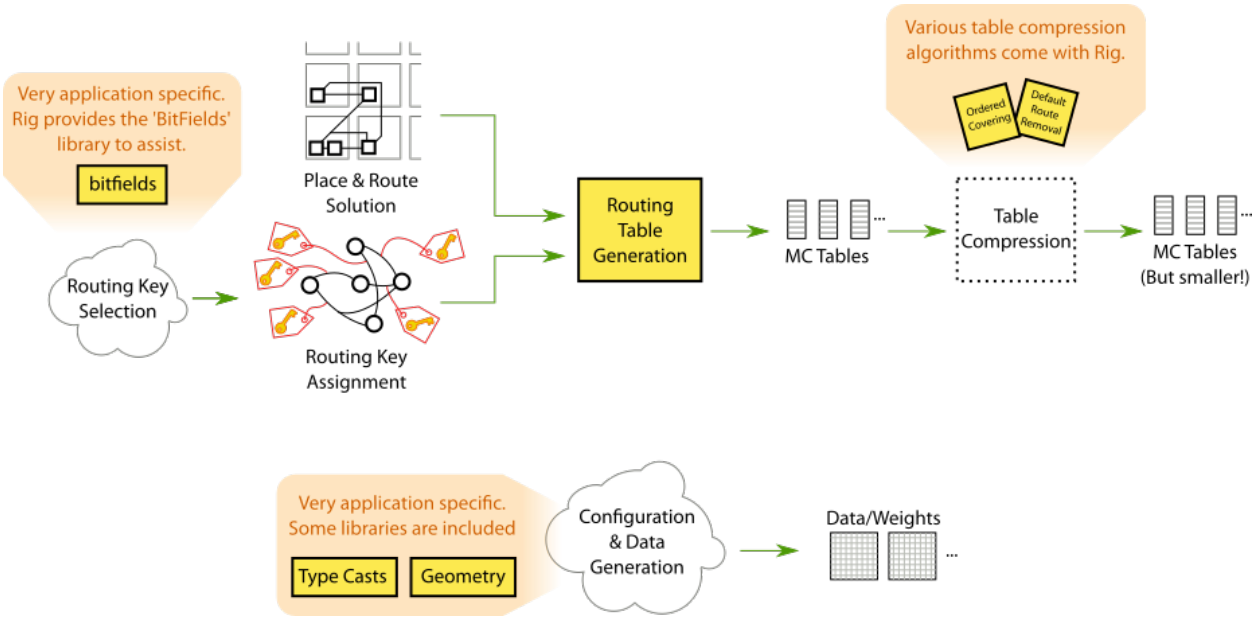
Most SpiNNaker applications provide a domain-specific API for describing whatever is to be simulated. This description is then *partitioned* into a graph of SpiNNaker-core-sized (*vertices*) which communicate with each other (via *nets*). This first step is usually very application specific and so Rig does not provide any functions to help.

In the place and route phase of application execution, the graph of vertices and nets is mapped onto specific cores and routes in SpiNNaker's network. Rig provides a number of utilities and algorithms for performing place and route tasks for SpiNNaker applications in the `rig.place_and_route` module. The figure below illustrates how Rig breaks place and route into three stages: placement, allocation and routing. In most applications this process can be handled automatically by a simple wrapper function but more advanced applications can customise the process.

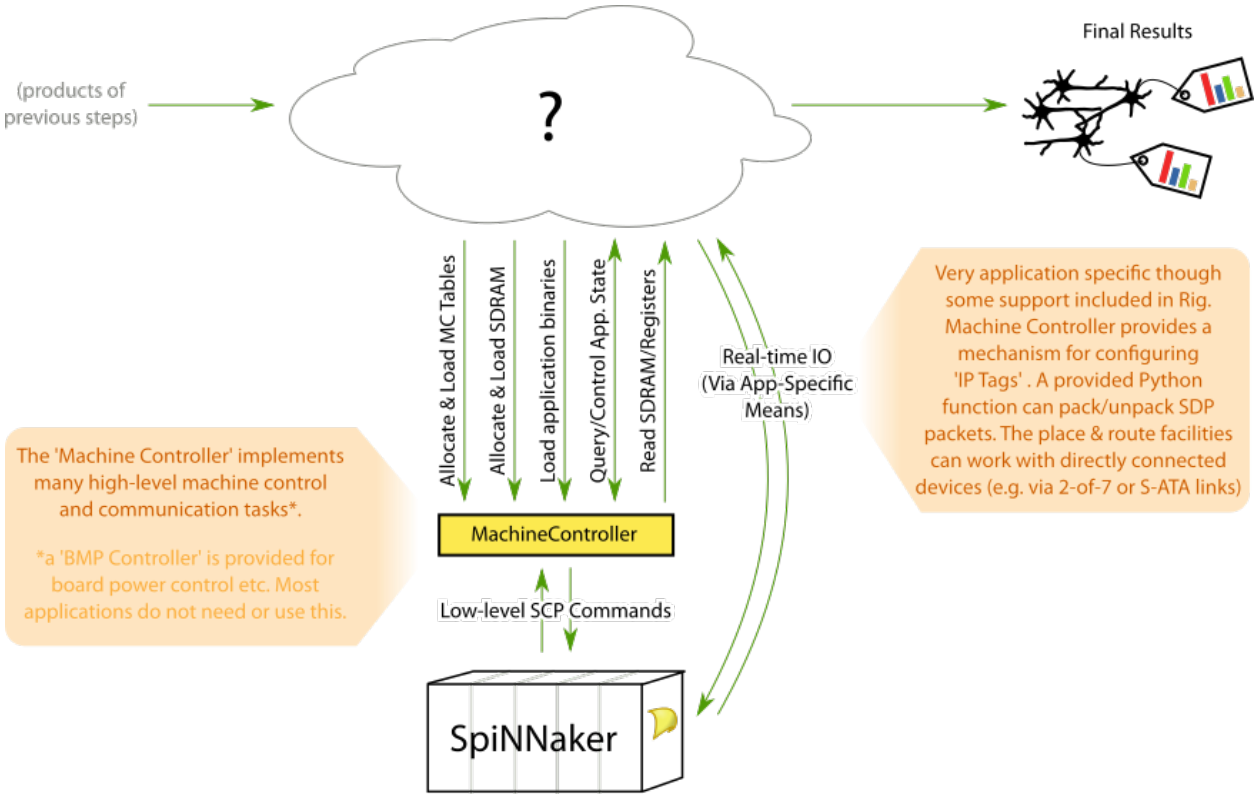
In the next phase of execution a typical application generates configuration data and routing tables which will be used by SpiNNaker application kernels (illustrated below). Though generating configuration data is highly application specific, Rig provides a limited set of libraries such as `rig.type_casts` for converting between floating and fixed point numbers, and `rig.geometry` for dealing with machine geometry. Rig also provides libraries to assist in selecting routing keys (`rig.bitfield`) and generating and compressing SpiNNaker routing tables (`rig.routing_table`).

In the final stage of execution (illustrated below), the SpiNNaker application kernels are loaded onto the SpiNNaker machine along with the generated configuration data and routing tables. Results are retrieved when computation completes. Rig provides a `MachineController` class which provides an efficient and high-level interface for


Various generic placers are included in Rig:
Simulated Annealing
Hilbert
Sequential
...
Separation of the allocation stage enables certain advanced place and route processes.
Greedy
Rig provides optimised fault-tolerant multicast routing algorithms.
NER
LDFR
(from partitioning)
SpiNNaker
Interrogation
MachineController
Vertices & Nets
Machine Description
Placer
Vertices mapped to chips
Allocator
Vertices mapped to specific cores
Router
Nets converted into routes (not routing tables)


Very application specific. Rig provides the 'BitFields' library to assist.
bitfields
Various table compression algorithms come with Rig.
Ordered Covering
Default Route Removal
Place & Route Solution
Routing Key Selection
Routing Key Assignment
Routing Table Generation
MC Tables
Table Compression
MC Tables (But smaller!)
Very application specific. Some libraries are included
Type Casts
Geometry
Configuration & Data Generation
Data/Weights

controlling and interacting with a SpiNNaker machine. Development of custom, real-time interaction with running SpiNNaker application kernels is also supported but remains a largely application specific task.



Note: It is important to note that the Rig library does not assume or rely on this particular program structure. Rig can be (and is) used in numerous applications with widly different structures. Examples include interactive use in the Python interpreter, cabling validation software and site-wide SpiNNaker job management and machine sharing.

You will need...

Before you start this tutorial assumes you have the following set-up and working:

1. An ARM cross-compiler. In this tutorial we presume GCC which can be downloaded and installed from Launchpad or from the package managers of good Linux distributions.
2. The latest version of the 'spinnaker_tools' low-level software package which can be downloaded from the SpiNNaker web pages.
3. A SpiNNaker machine. Any size from a 4-chip SpiNN-2 or SpiNN-3 board upwards will do. To make sure everything is working correctly, make sure you can get the example 'hello' app supplied with 'spinnaker_tools' to compile and run (see the 'spinnaker_tools' README).
4. A working Python 2 or Python 3 environment with Rig *installed*.

Without further delay, proceed to *part 01*.

1.2.2 01: Hello World

In this classic example we make a SpiNNaker application which simply prints "Hello, world!" on one core and then exits.

The source files used in this tutorial can be downloaded below:

- **Host program**
  - `hello.py`
- **SpiNNaker kernel**
  - `hello.c`
  - `Makefile`

As is traditional, our first application will simply print 'Hello, world!' and exit. In this example our SpiNNaker application kernel will simply write its greeting into memory on a SpiNNaker chip and then terminate. Our host program will:

- Load the application kernel
- Instruct SpiNNaker to run it
- Wait for the kernel to terminate
- Retrieve the message from SpiNNaker's memory and print it
- Clean up and quit

SpiNNaker Application Kernel

We start by writing the SpiNNaker application kernel itself which consists of a single call to `io_printf` in `hello.c`.

```
#include "sark.h"
void c_main(void)
{
  io_printf(IO_BUF, "Hello, world!\n");
}
```

This call writes our famous message to the "IO buffer", an area of system memory in each SpiNNaker chip which we can later read back from the host.

To compile our application we can use the standard two-line makefile:

```
APP := hello
include $(SPINN_DIRS)/make/app.make
```

To produce a compiled `hello.aplx` file ready for loading onto SpiNNaker, simply type:

```
$ make
```

Note: This makefile presumes your shell environment is set up correctly to use the 'spinnaker_tools'. This can be done by running:

```
$ source /path/to/spinnaker_tools/setup
```

Host-side application

Now that we have our compiled binary we must boot our SpiNNaker machine, load the application onto a core and then read back the IO buffer. We *could* do this using the *ybug* tool included with 'spinnaker_tools' but since we're building toward a real application we'll write a Python program which will automate all these steps.

Note: Even though we'll be writing our host programs in Python without using 'ybug', the 'ybug' tool remains a very useful debugging aid during development can can be safely used alongside your host application.

In our host program we'll use a part of the 'Rig' library called *MachineController* which provides a high-level interface for communicating with and controlling SpiNNaker machines. The first step in our program is to create an instance of the *MachineController* class to communicate with our SpiNNaker board:

```
import sys
from rig.machine_control import MachineController
mc = MachineController(sys.argv[1])
```

Note that we take the hostname/IP of the board as a command-line argument to avoid hard-coding it into our script.

To boot the machine we use the *boot()* method. If the machine is already booted, this command does nothing.

```
mc.boot()
```

Next we'll load our application using the *load_application()* method. This method loads our application onto core 1 of chip (0, 0), checks it was loaded successfully and then starts the program executing.

```
mc.load_application("hello.aplx", {(0, 0): {1}})
```

Note: *load_application()* can load an application onto many cores on many chips at once, hence the slightly unusual syntax.

When a SpiNNaker application kernel's `c_main` function returns, the application goes into the `exit` state. By using *wait_for_cores_to_reach_state()* we can wait for our hello world application to finish executing.

```
mc.wait_for_cores_to_reach_state("exit", 1)
```

After our application has exited we can fetch and print out the contents of the IO buffer to see the message printed by the application kernel. The buffer can be read using *get_iobuf()*. By convention Rig uses the name `p` – for processor – when identifying cores.

```
print(mc.get_iobuf(x=0, y=0, p=1))
```

As a final step we must send the "stop" signal to SpiNNaker using *send_signal()*. This frees up any resources allocated during the running of our application.

```
mc.send_signal("stop")
```

Running our application

Our script is now finished and can then be executed like so:

```
$ python hello.py BOARD_IP_HERE
Hello, world!
```

Note: The *boot()* command can take a few seconds to complete if the machine is not already booted. If the machine is already booted, the script should run almost instantaneously.

Once the excitement of being greeted by a super computer has worn off, it's time to set SpiNNaker to work on some 'real' computation. Let's head onward to *part 02*.

1.2.3 02: Reading and Writing SDRAM

Most interesting SpiNNaker application kernels require some sort of configuration data, and produce result data, which must be loaded and read back from the machine before and after executing respectively. As a result, a typical host program will:

- Allocate some memory on any SpiNNaker chips where an application kernel is to be loaded
- Write configuration data into this memory
- Load and run the application kernel
- Read and process the result data written into memory by the kernel

To illustrate this process we'll make a SpiNNaker application kernel which reads pair of 32-bit integers from memory, adds them together, stores the result back into memory and exits.

Much of the code in this example is unchanged from the previous example so we will only discuss the changes.

The source files used in this tutorial can be downloaded below:

- **Host program**
  - `adder.py`
- **SpiNNaker kernel**
  - `adder.c`
  - `Makefile`

Allocating SDRAM from the host

In our application, as in most real world applications, we'll use the on-chip SDRAM (shared between all cores on a chip) to load our two integers and store the result. By convention, the host program is responsible for allocating space in SDRAM.

The Rig *MachineController* class provides an *sdram_alloc()* method which we'll use to allocate 12 bytes of SDRAM on a SpiNNaker chip. In this example we'll allocate some SDRAM on chip (0, 0). The first 8 bytes will contain the two numbers to be summed and will be written by our host program. The last four bytes will be written by the SpiNNaker application kernel and will contain the resulting sum.

```
sdram_addr = mc.sdram_alloc(12, x=0, y=0, tag=1)
```

The *sdram_alloc()* method returns the address of a block of SDRAM on chip (0, 0) which was allocated.

We also need to somehow inform the SpiNNaker application kernel of this address. To do this we can use the 'tag' argument to identify the allocated memory block. Later, once the application kernel has been loaded it can use `sark_tag_ptr()` to discover the address of tagged SDRAM blocks. In most applications, memory used by an application running on core 1 is given tag 1, memory for core 2 is given tag 2 and so on. Since an application kernel can discover the core number it is running on using `spin1_get_core_id()`, the following line gets a pointer to the SDRAM block allocated for a particular core's application.

```
  uint32_t *numbers = sark_tag_ptr(spin1_get_core_id(), 0);
```

Note: Tags are assigned for a single SpiNNaker chip: tag numbers can be re-used on other chips.

Writing SDRAM from the host

After allocating our block of SDRAM we must populate it with the numbers to be added together. In this example, we pick two random numbers and, using Python's `struct` module, pack them into 8 bytes.

```
num_a = random.getrandbits(30)
num_b = random.getrandbits(30)
data = struct.pack("<II", num_a, num_b)
```

Note: The '<' prefix *must* be included in the struct format string to indicate that the data should be arranged in the little-endian order used by SpiNNaker.

The *write()* method of the *MachineController* is then used to write this value into the first 8 bytes of the SDRAM block we allocated.

```
mc.write(sdram_addr, data, x=0, y=0)
```

Warning: The *write()* method will attempt to perform any write you specify. Due caution should be used to avoid data corruption or illegal memory accesses.

Running the application kernel

With the SDRAM allocated, tagged and populated with data, we can load our application kernel as in the previous example using *load_application()*.

The application kernel adds together the numbers at the memory address discovered by `sark_tag_ptr()`, writes the result into memory and exits:

```
  numbers[2] = numbers[0] + numbers[1];
```

Note: Although SDRAM *can* be accessed directly like this, 'real' application kernels often use DMA requests to avoid blocking on slow memory access.

Reading and writing SDRAM from the host

After waiting for the application kernel to exit, the host can read the answer back using *read()* and unpack it using Python's `struct` module.

```
result_data = mc.read(sdram_addr + 8, 4, x=0, y=0)
result, = struct.unpack("<I", result_data)
print("{} + {} = {}".format(num_a, num_b, result))
```

As before, the last step is to send a "stop" signal to SpiNNaker using *send_signal()*. This signal will automatically free all allocated blocks of SDRAM.

In this tutorial we used some fairly low-level APIs for accessing SpiNNaker's memory. In the next tutorial we'll use some of Rig's higher-level APIs to make the process of accessing SpiNNaker's memory and cleaning up after an application easier and safer. Continue to *part 03*.

1.2.4 03: Reading and Writing SDRAM - Improved

We're now going to re-write the host-program from our previous example program, which used SpiNNaker to add two numbers together. In particular, some higher-level facilities of the *MachineController* will be used to make the host application simpler and more robust. The SpiNNaker application kernel, however, will remain unchanged.

The source files used in this tutorial can be downloaded below:

- **Host program**
  - `adder_improved.py`
- **SpiNNaker kernel (unchanged from *part 02*)**
  - `adder.c`
  - `Makefile`

Reliably stopping applications

Now that we're starting to allocate machine resources and write more complex programs it is important to be sure that the `stop` signal is sent to the machine at the end of our host application's execution. Rather than inserting a call to *send_signal()* into every exit code path, Rig provides the *application()* context manager which automatically sends a stop signal when the block ends:

```
with mc.application():
    # ...Application code...
```

When execution leaves an *application()* block, whether by reaching the end of the block, returning early from the function which contains it or encountering an exception, the `stop` signal is sent automatically.

In our new host program, we surround our application logic with an *application()* block. The *boot()* command is purposely placed outside the block since if the boot process fails, it is neither necessary nor possible to send a `stop` signal.

File-like memory access

When working with SDRAM it can be easy to accidentally access memory outside the range of an allocated buffer. To provide safer and more convenient access to SDRAM the *sdram_alloc_as_filelike()* method produces a file-like *MemoryIO* object for the allocated memory. This object can be used just like a conventional file, for example using *read()*, *write()* and *seek()* methods. All writes and reads to the file are automatically constrained to the allocated block of SDRAM preventing accidental corruption of memory. Additionally, users of an allocated block of memory need not know anything about the chip or address of the memory and may even be oblivious to the fact that they're using anything other than a normal file. This can simplify application code by avoiding the need to pass around additional information.

We replace the previous calls to *sdram_alloc()*, *write()* and *read()* with:

```
sdram = mc.sdram_alloc_as_filelike(12, x=0, y=0, tag=1)
sdram.write(data)
result_data = sdram.read(4)
```

Just like files, reads and writes occur immediately after the data the previous read and write encountered. *seek()* must be used move the 'read head' to other locations in memory. Note that in this case since the result value is written immediately after the two input values we do not need to seek before reading.

In the next part of the tutorial we'll use what we've learnt to take our first steps towards building a real application: a digital circuit simulator. Onward to *part 04*!

1.2.5 04: Circuit Simulation Proof-of-Concept

In this part of the tutorial we'll finally begin work on a real program: a digital circuit simulator. In this stage of the tutorial we'll build the SpiNNaker application kernels and a proof-of-concept host program to hook these kernels together in a fixed circuit to demonstrate everything working.

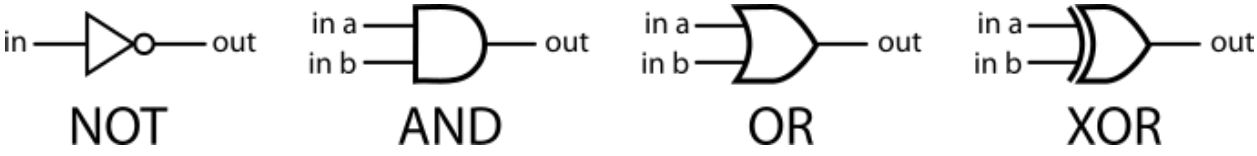
The source files used in this tutorial can be downloaded below:

- **Host program**
  - `circuit_simulator_proof.py`
- **SpiNNaker kernels**
  - `gate.c`
  - `stimulus.c`
  - `probe.c`
  - `Makefile`

Digital circuit simulation

In this tutorial we'll build a digital circuit simulator which (rather inefficiently) simulates the behaviour of a circuit made up of simple logic gates all wired together.

In our simulator, a logic gate is a device with one or two binary inputs and one binary output. In the picture below, four example logic gates are given along with truth-tables defining their behaviour.



Four simple 'logic gates' which each compute a simple boolean function. The 'truth tables' below enumerate the output values of each gate for every possible input.

NOT

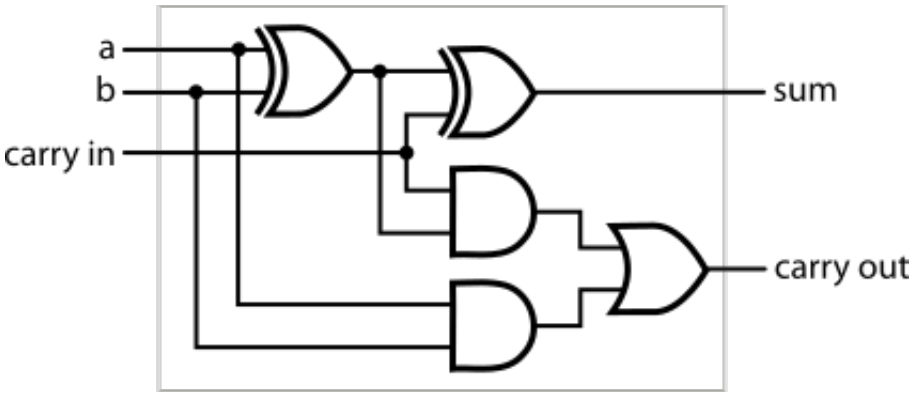
in	out
0	1
1	0

AND, OR and XOR

in a	in b	out		
		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Though on their own these logic gates don't do anything especially interesting by combining them into circuits more interesting behaviour can be achieved. Indeed, computer processors are little more than a carefully chosen collection of logic gates!

As an example, the circuit below is known as a 'full adder' which takes three one-bit binary numbers, 'a', 'b' and 'carry in' and adds them together to give a two-bit result whose least significant bit is 'sum' and whose most significant bit is 'carry out'.



For example if we set 'a' and 'carry in' to 1 and set 'b' to 0, the full-adder circuit will produce a 0 on its 'sum' output and a 1 on its 'carry out' output. Our inputs here are asking the full adder to compute "1 + 0 + 1" to which it dutifully answers "10" ("2" in binary).

Try following how the input values in this example flow through the full adder to produce the outputs in this example. For this tutorial it is not important to understand *why* the adder circuit works but you should be able to understand how input values flow through the circuit eventually resulting in outputs. Working out how to build a functioning CPU out of these gates is left as an exercise for the easily distracted reader and is well outside the scope of this tutorial...

Modelling a logic gate

Our circuit simulator will use a whole SpiNNaker core for each logic gate it simulates. Every millisecond each application core will recompute its output and send a multicast packet to any connected gates. When a gate receives a multicast packet indicating the value of one of its inputs it stores it to use next time the gate's output value is computed.

Rather than writing an individual SpiNNaker application kernel for each type of gate we want to simulate, we'll instead write a single application kernel which is configured with a look-up table (i.e. a 'truth table') by the host to whatever any functions we require.

`gate.c` contains the full source listing for our gate kernel. We'll walk through the key parts below.

```
void on_tick(uint32_t ticks, uint32_t arg1) {
        // Terminate after the specified number of ticks.
        // NB: the tick count starts from 1!
```

(continues on next page)

(continued from previous page)

```
        if (ticks > config->sim_length) {
                spin1_exit(0);
                return;
        }

        // Look-up the new output value
        uint32_t lut_bit_number = last_input_a | (last_input_b << 1);
        uint32_t output = (config->lut >> lut_bit_number) & 1;

        // Send the output value of the simulated gate as the payload in a
        // multicast packet.
        spin1_send_mc_packet(config->output_key, output, WITH_PAYLOAD);
}
```

The timer is configured to call the `on_tick()` function every millisecond. This function looks-up the desired output value in a lookup table based on the most recently received input values. The output value is then sent via a SpiNNaker multicast packet. The function is also responsible for terminating the simulation after a predetermined amount of time.

The `last_input_a` and `last_input_b` variables are set by the `on_mc_packet()` function which is called whenever a multicast packet arrives at the core.

```
void on_mc_packet(uint32_t key, uint32_t payload) {
        if (key == config->input_a_key)
                last_input_a = payload;
        if (key == config->input_b_key)
                last_input_b = payload;
}
```

This function simply checks to see which input the incoming multicast packet is related to by checking its key against the expected key for each of the two inputs.

The `config` struct used by the two callback functions above is expected to be written by the host and contains several fields describing the desired behaviour of the gate being simulated.

```
typedef struct {
        // The number of milliseconds to run for
        uint32_t sim_length;

        // The routing key used by multicast packets relating to input a
        uint32_t input_a_key;

        // The routing key used by multicast packets relating to input b
        uint32_t input_b_key;

        // The routing key to use when transmitting the output value
        uint32_t output_key;

        // A lookup table from input a and b to output value.
        //
        // =======  =======  ==============
        // input a  input b  lut bit number
        // =======  =======  ==============
        // 0        0        0
        // 1        0        1
        // 0        1        2
        // 1        1        3
        // =======  =======  ==============
```

(continues on next page)

(continued from previous page)

```
        uint32_t lut;
} config_t;

config_t *config;
```

The pointer to the `config` struct is set using the `sark_tag_ptr()` as described in the previous tutorials and the callbacks setup in the `c_main()` function.

Stimulus and probing kernels

To make our simulator useful we need to be able to provide input stimulus and record the output produced. To do this we'll create two additional SpiNNaker application kernels: `stimulus.c` and `probe.c`.

The stimulus kernel will simply output a sequence of values stored in memory, one each millisecond. As in the gate kernel, a configuration struct is defined which the host is expected to populate:

```
typedef struct {
        // The number of milliseconds to run for
        uint32_t sim_length;

        // The routing key to use when transmitting the output value
        uint32_t output_key;

        // An array of ceil(sim_length/8) bytes where bit-0 of byte[0] contains the␣
↪first
        // bit to send, bit-1 gives the second bit and bit-0 of byte[1] gives the
        // eighth bit and so on...
        uint8_t stimulus[];
} config_t;

config_t *config;
```

This configuration is then used by the timer interrupt to send output values into the network:

```
void on_tick(uint32_t ticks, uint32_t arg1) {
        // The tick count provided by Spin1 API starts from 1 so decrement to get a
        // 0-indexed count.
        ticks--;

        // Terminate after the specified number of ticks.
        if (ticks >= config->sim_length) {
                spin1_exit(0);
                return;
        }

        // Get the next output value
        uint32_t output = (config->stimulus[ticks / 8] >> (ticks % 8)) & 1;

        // Send the new output value as the payload in a multicast packet.
        spin1_send_mc_packet(config->output_key, output, WITH_PAYLOAD);
}
```

The probe kernel takes on the reverse role: every millisecond it records into memory the most recent input value it received. The host can later read this data back. Once more, a configuration struct is defined which the host will populate and to which the probe will add recorded data:

```
typedef struct {
        // The number of milliseconds to run for
        uint32_t sim_length;

        // The routing key used by multicast packets relating to the probed input
        uint32_t input_key;

        // An array of ceil(sim_length/8) bytes where bit-0 of byte[0] will be
        // written with value in the first millisecond, bit-1 gives the value in the
        // second millisecond and bit-0 of byte[1] gives the value in the eighth
        // millisecond and so on...
        uchar recording[];
} config_t;

config_t *config;
```

The 'recording' array is zeroed during kernel startup to save the host from having to write the zeroes over the network:

```
        for (int i = 0; i < (config->sim_length + 7)/8; i++)
                config->recording[i] = 0;
```

The array is then written to once per millisecond with the most recently received value:

```
void on_tick(uint32_t ticks, uint32_t arg1) {
        // The tick count provided by Spin1 API starts from 1 so decrement to get a
        // 0-indexed count.
        ticks--;

        // Terminate after the specified number of ticks.
        if (ticks >= config->sim_length) {
                spin1_exit(0);
                return;
        }

        // Pause for a while to allow values sent during this millisecond to arrive
        // at this core.
        spin1_delay_us(700);

        // Record the most recently received value into memory
        config->recording[ticks/8] |= last_input << (ticks % 8);
}
```

As in the gate kernel, a callback on multicast packet arrival keeps track of the most recently received input value:

```
void on_mc_packet(uint32_t key, uint32_t payload) {
        if (key == config->input_key)
                last_input = payload;
}
```

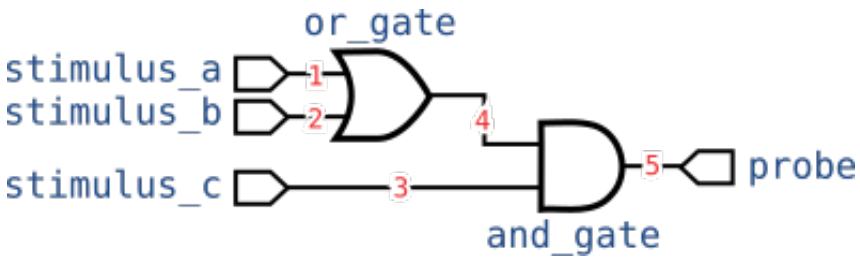
Compiling the kernels

A `Makefile` is provided which builds all three kernels when you type:

```
$ make
```

A proof-of-concept host program

To try out our new application kernels we'll now put together a proof-of-concept host application which uses our kernels to simulate a single circuit and hard-codes all configuration data for each application kernel. Our proof-of-concept system will simulate the following circuit:

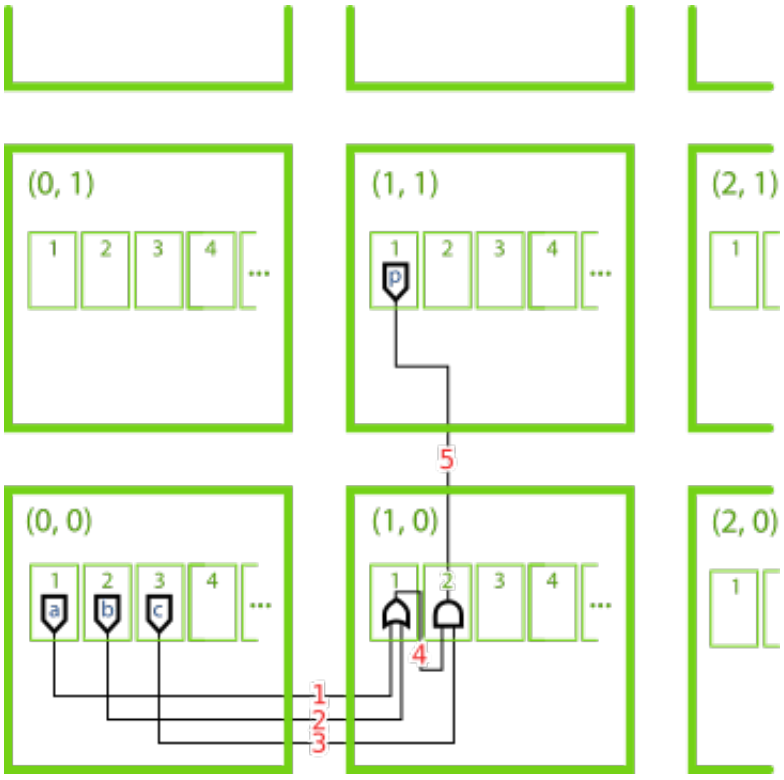


To do this we'll need 6 SpiNNaker cores (one for each gate, probe and stimulus). We'll arbitrarily use the following assignment of cores:

Component	Chip	Core	Kernel
`stimulus_a`	(0, 0)	1	`stimulus`
`stimulus_b`	(0, 0)	2	`stimulus`
`stimulus_c`	(0, 0)	3	`stimulus`
`or_gate`	(1, 0)	1	`gate`
`and_gate`	(1, 0)	2	`gate`
`probe`	(1, 1)	1	`probe`

The five wires numbered 1 to 5 in the circuit diagram will be carried by multicast routes whose key is the wire number.

This assignment is illustrated in the figure below:



Note: Core 0 every SpiNNaker chip (not shown in the figure above) is always used by the 'monitor process' (SC&MP) which is used to facilitate control of the system. It cannot be used to run SpiNNaker application kernels.

We'll run our simulation for 64 ms and configure each stimulus kernel such that each of the 8 combinations of stimulus value are held for 8 ms each to allow time for the signals to propagate through the circuit.

The proof-of-concept host program is provided in full in `circuit_simulator_proof.py` and we'll walk through the key steps below. After creating a *MachineController* instance and booting the machine as usual, the first step is to allocate memory for the configuration structs of each of the six applications using *sdram_alloc_as_filelike()*:

```
with mc.application():
    # Allocate a tagged block of SDRAM to hold the configuration struct for
    # each application kernel.
    with mc(x=0, y=0):
        # Space for sim_length, output_key and space for 64 ms of stimulus
        # data.
        stimulus_a_config = mc.sdram_alloc_as_filelike(4 + 4 + 8, tag=1)
        stimulus_b_config = mc.sdram_alloc_as_filelike(4 + 4 + 8, tag=2)
        stimulus_c_config = mc.sdram_alloc_as_filelike(4 + 4 + 8, tag=3)

    with mc(x=1, y=0):
        # Space for all 5 uint32_t values in the config struct
        or_gate_config = mc.sdram_alloc_as_filelike(5 * 4, tag=1)
        and_gate_config = mc.sdram_alloc_as_filelike(5 * 4, tag=2)

    # Space for sim_length, input_key and space for 64 ms of stimulus data.
    probe_config = mc.sdram_alloc_as_filelike(4 + 4 + 8, x=1, y=1, tag=1)
```

Tip: In the code above, *MachineController* is used as a context manager. This allows a common set of arguments to be specified once, the chip coordinate arguments in this example, and omitted in subsequent calls.

Next we write the configuration structs using Python's `struct` module and the bitarray package to pack the desired data:

```
    # The stimulus data (tries every combination of a, b and c for 8 ms each)
    #          |       |       |       |       |       |       |       |       |
    stim_a = "0000000011111111000000001111111100000000111111110000000011111111"
    stim_b = "0000000000000000111111111111111100000000000000001111111111111111"
    stim_c = "0000000000000000000000000000000011111111111111111111111111111111"

    # Write stimulus configuration structs
    stimulus_a_config.write(struct.pack("<II", 64, 0x00000001))
    stimulus_a_config.write(bitarray(stim_a, endian="little").tobytes())

    stimulus_b_config.write(struct.pack("<II", 64, 0x00000002))
    stimulus_b_config.write(bitarray(stim_b, endian="little").tobytes())

    stimulus_c_config.write(struct.pack("<II", 64, 0x00000003))
    stimulus_c_config.write(bitarray(stim_c, endian="little").tobytes())

    # Write gate configuration structs, setting the look-up-tables to implement
    # the two gates' respective functions.
    or_gate_config.write(struct.pack("<5I",
```

(continues on next page)

(continued from previous page)

```
                                    64,          # sim_length
                                    0x00000001,  # input_a_key
                                    0x00000002,  # input_b_key
                                    0x00000004,  # output_key
                                    0b1110))     # lut (OR)
    and_gate_config.write(struct.pack("<5I",
                                     64,          # sim_length
                                     0x00000004,  # input_a_key
                                     0x00000003,  # input_b_key
                                     0x00000005,  # output_key
                                     0b1000))     # lut (AND)

    # Write the probe's configuration struct (note this doesn't write to the
    # buffer used to store recorded values).
    probe_config.write(struct.pack("<II", 64, 0x00000005))
```

In order to route the multicast packets to their appropriate destinations we must define some routing table entries on the chips we're using. We build up a dictionary which contains a list of *RoutingTableEntry* tuples for each SpiNNaker chip where routing table entries must be added. A *RoutingTableEntry* tuple corresponds directly to a SpiNNaker routing table entry which routes packets to the supplied `set` of *Routes* when they match the supplied key and mask value.

The routing tables described are finally loaded onto their respective chips using the *load_routing_tables()* method of the *MachineController*.

Note: The details of SpiNNaker's multicast router are outside of the scope of this tutorial. In the next part of the tutorial we'll use Rig's place-and-route facilities to generate these tables automatically so understanding how SpiNNaker's router works is not strictly necessary (though often helpful!).

In brief: we must add a routing entry wherever a packet enters the network, changes direction or leaves the network for a local core. A packet's routing key is matched by an entry in the table whenever `(packet_key & table_entry_mask) == table_entry_key`. If no routing entry matches a packet's key, the packet is 'default-routed' in a straight line to the opposite link to the one it arrived on. The Section 10.4 (page 39) of the SpiNNaker Datasheet provides a good introduction to SpiNNaker's multicast router and routing tables.

```
    # Define routing tables for each chip
    routing_tables = {(0, 0): [],
                      (1, 0): [],
                      (1, 1): []}

    # Wire 1
    routing_tables[(0, 0)].append(
        RoutingTableEntry({Routes.east}, 0x00000001, 0xFFFFFFFF))
    routing_tables[(1, 0)].append(
        RoutingTableEntry({Routes.core_1}, 0x00000001, 0xFFFFFFFF))

    # Wire 2
    routing_tables[(0, 0)].append(
        RoutingTableEntry({Routes.east}, 0x00000002, 0xFFFFFFFF))
    routing_tables[(1, 0)].append(
        RoutingTableEntry({Routes.core_1}, 0x00000002, 0xFFFFFFFF))

    # Wire 3
    routing_tables[(0, 0)].append(
```

(continues on next page)

(continued from previous page)

```
        RoutingTableEntry({Routes.east}, 0x00000003, 0xFFFFFFFF))
    routing_tables[(1, 0)].append(
        RoutingTableEntry({Routes.core_2}, 0x00000003, 0xFFFFFFFF))

    # Wire 4
    routing_tables[(1, 0)].append(
        RoutingTableEntry({Routes.core_2}, 0x00000004, 0xFFFFFFFF))

    # Wire 5
    routing_tables[(1, 0)].append(
        RoutingTableEntry({Routes.north}, 0x00000005, 0xFFFFFFFF))
    routing_tables[(1, 1)].append(
        RoutingTableEntry({Routes.core_1}, 0x00000005, 0xFFFFFFFF))

    # Allocate and load the above routing entries onto their respective chips
    mc.load_routing_tables(routing_tables)
```

We're finally ready to load our application kernels onto their respective chips and cores using *load_application()*:

```
    mc.load_application({
        "stimulus.aplx": {(0, 0): {1, 2, 3}},
        "gate.aplx": {(1, 0): {1, 2}},
        "probe.aplx": {(1, 1): {1}},
    })
```

Note: *load_application()* uses an efficient flood-fill mechanism to applications onto several chips and cores simultaneously.

The 'Spin1 API' used to write the application kernels causes our kernels to wait at the 'sync0' barrier once the `spin1_start()` function is called at the end of `c_main()`. We will use *wait_for_cores_to_reach_state()* to wait for all six application kernels to reach the 'sync0' barrier:

```
    mc.wait_for_cores_to_reach_state("sync0", 6)
```

Next we send the 'sync0' signal using *send_signal()*. This starts our application kernels running. After 64 ms all of the applications should terminate and we wait for them to exit using *wait_for_cores_to_reach_state()*.

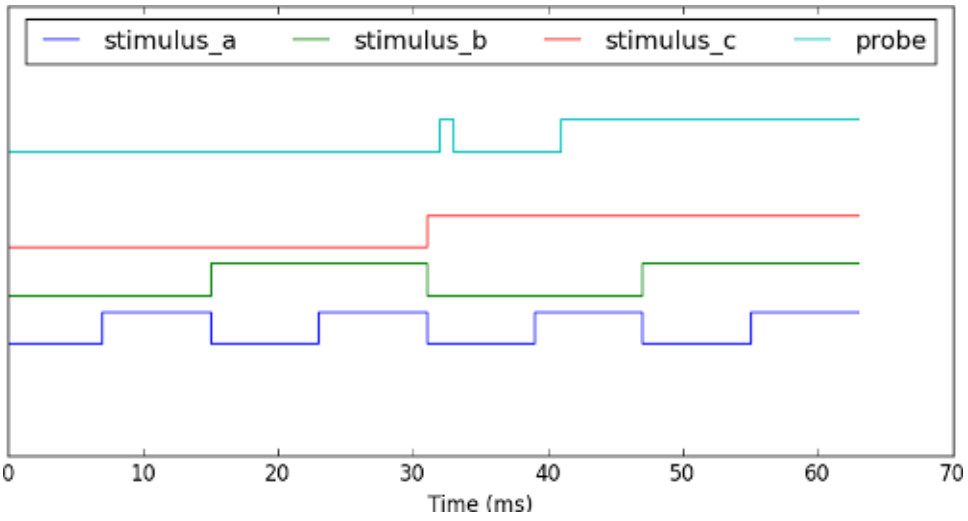
```
    mc.send_signal("sync0")
    time.sleep(0.064)  # 64 ms
    mc.wait_for_cores_to_reach_state("exit", 6)
```

Next we retrieve the result data recorded by the probe:

```
    probe_recording = bitarray(endian="little")
    probe_recording.frombytes(probe_config.read(8))
```

Note: Note that the `probe_config` file-like object's read pointer was moved to the start of the `recording` array when the configuration data before it was written earlier in the host program.

The stimulus and recorded data are plotted using `pyplot` to produce a figure like the one below:



Note: The recording shows a 'glitch' at time=32 which is caused by propagation delays in our circuit rather than a bug in our simulator. In fact, our simulator has accurately modelled an unintended feature of our circuit!

And there we have it: a real digital circuit simulation running on SpiNNaker! Unfortunately, our simulator is not especially flexible. Changing the circuit requires re-writing swathes of code and hand-generating yet more routing tables. In the next part of the tutorial we'll make use of the automatic place-and-route features included in the Rig library to take care of this for us. So without further delay, let's proceed to *part 05*!

1.2.6 05: Circuit Simulation

In the *previous part of this tutorial* we built a simple digital circuit simulator using several application kernels running on multiple SpiNNaker chips which communicated with multicast packets. In our proof-of-concept host program, the chip and core to use for each kernel was chosen by hand and all routing tables were written manually. Though this works, it made our simulator incredibly inflexible and the host program hard to modify and extend.

In this part of the tutorial we'll leave the application kernels unchanged but re-write our host program to make use of the automatic place-and-route tools provided by Rig. These tools automate the process of assigning application kernels to specific cores and generating routing tables while attempting to make efficient use of the machine. We'll also restructure our host program to be more like a real-world application complete with a simple user-facing interface.

The source files used in this tutorial can be downloaded below:

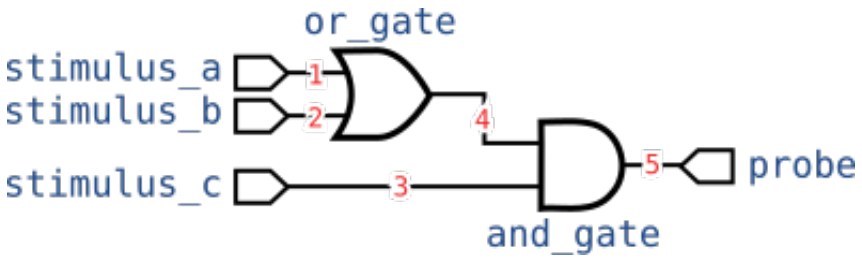
- **Host program**
  - `circuit_simulator.py`
- **Example circuit simulation script**
  - `example_circuit.py`
- **SpiNNaker kernels (unchanged from *part 04*)**
  - `gate.c`
  - `stimulus.c`
  - `probe.c`
  - `Makefile`

Defining the circuit simulator user interface/API

If our circuit simulator is to be useful it must present a sensible API to allow users to describe their circuits. In this example we'll implement an API which looks like this:

```
import sys

from circuit_simulator import Simulator, Stimulus, Or, And, Probe

# Define a 64 ms simulation to be run on the given SpiNNaker machine
sim = Simulator(sys.argv[1], 64)

# Define three stimulus generators which together produce all 8 combinations of
# values.
stimulus_a = Stimulus(
    sim, "0000000011111111000000001111111100000000111111110000000011111111")
stimulus_b = Stimulus(
    sim, "0000000000000000111111111111111100000000000000001111111111111111")
stimulus_c = Stimulus(
    sim, "0000000000000000000000000000000011111111111111111111111111111111")

# Define the two gates
or_gate = Or(sim)
and_gate = And(sim)

# Define a probe to record the output of the circuit
probe = Probe(sim)

# Wire everything together
or_gate.connect_input("a", stimulus_a.output)
or_gate.connect_input("b", stimulus_b.output)

and_gate.connect_input("a", stimulus_c.output)
and_gate.connect_input("b", or_gate.output)

probe.connect_input(and_gate.output)

# Run the simulation
sim.run()

# Print the results
print("Stimulus A: " + stimulus_a.stimulus)
print("Stimulus B: " + stimulus_b.stimulus)
print("Stimulus C: " + stimulus_c.stimulus)
print("Probe:      " + probe.recorded_data)
```

This script defines the same circuit which we hard-coded in *part 04*:



With our desired API in mind, lets design our circuit simulator!

Place and Route using Rig

Before diving into the code it is first important to understand what the Rig place-and-route tools do.

Rig provides a suite of placement and routing algorithms in its *rig.place_and_route* module. In essence, these algorithms accept abstract descriptions of graphs of communicating SpiNNaker application kernels as input. Based on this information the place and route algorithms select which core each kernel will be loaded onto, keeping communicating cores close together to reduce network load. In addition, routing tables which make efficient use of SpiNNaker's network are generated.

In Rig terminology, the abstract (hyper-)graph of application kernels are known as *vertices* which are connected together by *nets*:

vertices Approximately speaking, a vertex represents a group of cores and SDRAM which must be assigned in one piece to a chip somewhere. In our circuit simulator, a vertex represents a single gate, stimulus or probe and each requires a single core and some quantity of SDRAM.

nets A net typically represents a 1-to-many flow of multicast packets between vertices. A net has a single *source* vertex and many *sink* vertices. In our circuit simulator, a net corresponds to a wire in our circuit, where the source is the gate or stimulus output driving the wire and the sinks are the connected gate and probe inputs.

In addition to graph of vertices and nets, the place and route tools require a description of the SpiNNaker machine our simulation will be running on. As we will see later, the *MachineController* provides a method for gathering this information.

Building the circuit simulator API

What follows is a (non-linear) walk-through of the most important parts of the circuit simulator host program provided in `circuit_simulator.py`.

In most host applications built with Rig, the graph of vertices and nets fed to the place and route tools are generated from application-specific data structures shortly before performing the place-and-route. This allows the majority of the application to use data structures which best fit the application. In this circuit simulator example we'll follow this approach too, so let's start by defining the Python classes which make up the API.

Defining a wire

A wire represents a connection from one the output of one component to the inputs of many other components and is defined as follows:

```
class _Wire(object):
    """A wire which connects one component's output to many components'
    inputs.

    For internal use: to be constructed via :py:meth:`.Simulator._new_wire`
    only.
    """

    def __init__(self, source, sinks, routing_key):
        """Defines a new wire from source to sinks which will use the specified
        routing key.

        Parameters
        ----------
        source : component
        sinks : [component, ...]
```

(continues on next page)

(continued from previous page)

```
        routing_key : int
        """
        self.source = source
        self.sinks = sinks
        self.routing_key = routing_key
```

A `_Wire` instance contains a source component, a `list` of sink components and a unique routing key to use in the simulation. The `Simulator` object (to be defined later) will be responsible for creating new `_Wire` objects.

Defining components (gates, stimuli and probes)

At the heart of our circuit simulator is our two-input, one-output, lookup-table-based logic gate so let's define our `Gate` component first like so:

```
class Gate(object):
    """A 2-input 1-output logic gate implemented using a lookup-table."""

    def __init__(self, simulator, lookup_table):
        """Define a new gate.

        Parameters
        ----------
        simulator : :py:class:`.Simulator`
            The simulator which will be responsible for simulating this gate.
        lookup_table : int
            A lookup table giving the output value of the gate as a 4-bit
            number where each bit gives the output for a particular combination
            of input values.

            =======  =======  ==============
            input a  input b  lut bit number
            =======  =======  ==============
            0        0        0
            1        0        1
            0        1        2
            1        1        3
            =======  =======  ==============
        """
        self._simulator = simulator
        self._lookup_table = lookup_table

        # Register this component with the simulator
        self._simulator._add_component(self)

        # The two inputs, initially not connected
        self._inputs = {"a": None, "b": None}

        # A new wire will be created and sourced by this gate
        self.output = self._simulator._new_wire(self)

    def connect_input(self, name, wire):
        """Connect the specified input to a wire."""
        self._inputs[name] = wire
        wire.sinks.append(self)
```

In the constructor we simply store a reference to the `Simulator` object along with a copy of the lookup table pro-

vided. We also inform the `Simulator` of the existance of the component using `Simulator._add_component`. The `_inputs` attribute will hold references to the `_Wires` connected to each input and the `output` attribute holds a reference to (a newly created) `_Wire` which will be driven by the gate.

The `Gate.connect_input` method connects a `_Wire` to an input by storing a reference to the `_Wire` object and adding the component to the `_Wire`'s list of sinks.

We also define various subclasses of `Gate` which, for the sake of convenience, simply define the lookup table to be used. For example an AND-gate component is defined like so:

```
class And(Gate):
    """An AND gate."""

    def __init__(self, simulator):
        super(And, self).__init__(simulator, 0b1000)
```

The `Probe` object is defined in a similar way to the `Gate` but doesn't define an output:

```
class Probe(object):
    """A 1-bit recording probe."""

    def __init__(self, simulator):
        """Define a new probe.

        Parameters
        ----------
        simulator : :py:class:`.Simulator`
            The simulator in which the probe will be used.
        """
        self._simulator = simulator
        self.recorded_data = None

        # Register this component with the simulator
        self._simulator._add_component(self)

        # The input, initially disconnected
        self._input = None

    def connect_input(self, wire):
        """Probe the specified wire."""
        self._input = wire
        wire.sinks.append(self)
```

Finally, the `Stimulus` object is defined but, since it doesn't have any inputs, the `connect_input` method is excluded:

```
class Stimulus(object):
    """A 1-bit stimulus source."""

    def __init__(self, simulator, stimulus=""):
        """Define a new stimulus source.

        Parameters
        ----------
        simulator : :py:class:`.Simulator`
            The simulator in which the stimulus will be used.
        stimulus : str
            A string of "0" and "1"s giving the stimulus to generate for each
```

(continues on next page)

(continued from previous page)

```
            millisecond in the simulation. Will be zero-padded or truncated to
            match the length of the simulation.
        """
        self._simulator = simulator
        self.stimulus = stimulus

        # Register this component with the simulator
        self._simulator._add_component(self)

        # A new wire will be created sourced by this stimulus generator
        self.output = self._simulator._new_wire(self)
```

Defining the simulator

All that remains to be defined of our API is the `Simulator` object. The `Simulator` simply stores the hostname and simulation length provided and maintains lists of components and wires which have been added to the simulation:

```
class Simulator(object):
    """A SpiNNaker-based digital logic simulator."""

    def __init__(self, hostname, length):
        """Create a new simulation.

        Parameters
        ----------
        hostname : str
            The hostname or IP of the SpiNNaker machine to use.
        length : int
            The number of milliseconds to run the simulation for.
        """
        self._hostname = hostname
        self.length = length

        # A list of components added to the simulation
        self._components = []

        # A list of wires used in the simulation
        self._wires = []

    def _add_component(self, component):
        """Add a component to the simulation.

        Called internally by components on construction.
        """
        self._components.append(component)

    def _new_wire(self, source, sinks=None):
        """Create a new :py:class:`._Wire` with a unique routing key."""
        # Assign sequential routing key to new nets.
        wire = _Wire(source, sinks if sinks is not None else [], len(self._wires))
        self._wires.append(wire)

        return wire
```

Making it work

At this point, our API is complete with the notable exception of the `Simulation.run()` method. At a high level, the `run()` method performs the following steps:

- Build a graph of the form accepted by Rig's place and route tools.
- Perform place and route.
- Load the configuration data, routing tables and application kernels required.
- Run the simulation.
- Read back results captured by probes.

We'll now proceed to break down this function and look at its operation in detail.

Building a place-and-routeable graph

To perform place and route we must build a graph describing our simulation in the format required by Rig.

The first thing we need to do is define the resources required by each vertex in the graph. Rig allows us to use any Python `object` to represent a vertex and since each component in our simulation will become a vertex in our graph we'll use the `object`s we defined above to identify the vertices. We build a `vertices_resources` dictionary which enumerates the resources consumed by each vertex in our application:

```
vertices_resources = {
    # Every component runs on exactly one core and consumes a certain
    # amount of SDRAM to hold configuration data.
    component: {Cores: 1, SDRAM: component._get_config_size()}
    for component in self._components
}
```

Each entry in the `vertices_resources` dictionary contains another dictionary mapping 'resources' to the required quantities of each resource. As in most applications, the only resources we care about are Cores and SDRAM. By convention these resources are identified to by the corresponding *`Cores`* and *`SDRAM`* sentinels defined by Rig.

Each vertex requires exactly one core but the amount of SDRAM required depends on the type of component and length of the simulation. A `_get_config_size()` method is added to each of our component types to compute their SDRAM requirements:

```
class Gate(object):
    def _get_config_size(self):
        """Get the size of configuration block needed for this gate."""
        # The config contains 5x uint32_t
        return 5 * 4
```

```
class Probe(object):
    def _get_config_size(self):
        """Get the size of configuration block needed for this probe."""
        # The config contains 2x uint32_t and a byte for every 8 bits of
        # recorded data.
        return (2 * 4) + ((self._simulator.length + 7) // 8)
```

```
class Stimulus(object):
    def _get_config_size(self):
        """Get the size of configuration block needed for this stimulus."""
        # The config contains 2x uint32_t and a byte for every 8 bits of
```

(continues on next page)

(continued from previous page)

```
        # stimulus data.
        return (2 * 4) + ((self._simulator.length + 7) // 8)
```

Next we must also define the filename of the spinnaker application kernel (i.e. the APLX file) used for each vertex.

```
vertices_applications = {component: component._get_kernel()
                         for component in self._components}
```

Once again we support this by adding a `_get_kernel()` method to each component type:

```
class Gate(object):
    def _get_kernel(self):
        """Get the filename of the SpiNNaker application kernel to use."""
        return "gate.aplx"
```

```
class Probe(object):
    def _get_kernel(self):
        """Get the filename of the SpiNNaker application kernel to use."""
        return "probe.aplx"
```

```
class Stimulus(object):
    def _get_kernel(self):
        """Get the filename of the SpiNNaker application kernel to use."""
        return "stimulus.aplx"
```

Next, we enumerate the nets representing the streams of multicast packets flowing between vertices, as well as the routing keys and masks used for each net. Rig expects nets to be defined by *Net* objects. Like the `_Wire` objects in our simulator, *Net*s simply contain a source vertex and a list of sink vertices. In the code below we build a `dict` mapping *Net*s to `(key, mask)` tuples for each wire in the simulation:

```
net_keys = {Net(wire.source, wire.sinks): (wire.routing_key,
                                           0xFFFFFFFF)
            for wire in self._wires}
nets = list(net_keys)
```

The final piece of information required is a description of the SpiNNaker machine onto which our application will be placed and routed. Using a *MachineController* we first *boot()* the machine and then interrogate it using *get_system_info()* which returns a `SystemInfo` object. This object contains a detailed description of the machine, for example, enumerating working cores and links. This description will be used shortly to perform place and route.

```
mc = MachineController(self._hostname)
mc.boot()
system_info = mc.get_system_info()
```

Place and route

The place and route process can be broken up into many steps such as placement, allocation, routing and routing table generation. Though some advanced applications may find it useful to break these steps apart, our circuit simulator, like many other applications, does not. Rig provides a *place_and_route_wrapper()* function which saves us from the 'boilerplate' of doing each step separately. This function takes the graph description we constructed above and performs the place and route process in its entirety.

```
placements, allocations, application_map, routing_tables = \
    place_and_route_wrapper(vertices_resources,
                            vertices_applications,
                            nets, net_keys,
                            system_info)
```

The `placements` and `allocations` `dict` returned by *`place_and_route_wrapper()`* together define the specific chip and core each vertex has been assigned to (see *`place()`* and *`allocate()`* for details).

`application_map` is a `dict` describing what application kernels need to be loaded onto what cores in the machine.

Finally, `routing_tables` contains a `dict` giving the routing tables to be loaded onto each core in the machine.

Loading and running the simulation

We are now ready to load and execute our circuit simulation on SpiNNaker. The first step is to allocate blocks of SDRAM containing configuration data on every chip where our application kernels will run.

The *`sdram_alloc_for_vertices()`* utility function takes a *`MachineController`* and the `placements` and `allocations` `dict`s produced during place and route and allocates a block of SDRAM for each vertex. Each allocation is given a tag matching the core number of the vertex, and the size of the allocation is determined by the quantity of *`SDRAM`* consumed by the vertex, as originally indicated in `vertices_resources`.

```
memory_allocations = sdram_alloc_for_vertices(mc, placements,
                                              allocations)
```

The `dict` returned is a mapping from each vertex (i.e. instances of our component classes) to a *`MemoryIO`* file-like interface to SpiNNaker's memory.

We add a `_write_config` method to each of our component classes which is passed a *`MemoryIO`* object into which configuration data is written.

```
for component, memory in memory_allocations.items():
    component._write_config(memory)
```

The `_write_config` functions for each component type are as follows:

```
class Gate(object):
    def _write_config(self, memory):
        """Write the configuration for this gate to memory."""
        memory.seek(0)
        memory.write(struct.pack("<5I",
                                 # sim_length
                                 self._simulator.length,
                                 # input_a_key
                                 self._inputs["a"].routing_key
                                 if self._inputs["a"] is not None
                                 else 0xFFFFFFFF,
                                 # input_b_key
                                 self._inputs["b"].routing_key
                                 if self._inputs["b"] is not None
                                 else 0xFFFFFFFF,
                                 # output_key
                                 self.output.routing_key,
                                 # lut
                                 self._lookup_table))
```

```
class Probe(object):
    def _write_config(self, memory):
        """Write the configuration for this probe to memory."""
        memory.seek(0)
        memory.write(struct.pack("<II",
                                 # sim_length
                                 self._simulator.length,
                                 # input_key
                                 self._input.routing_key
                                 if self._input is not None
                                 else 0xFFFFFFFF))
```

```
class Stimulus(object):
    def _write_config(self, memory):
        """Write the configuration for this stimulus to memory."""
        memory.seek(0)
        memory.write(struct.pack("<II",
                                 # sim_length
                                 self._simulator.length,
                                 # output_key
                                 self.output.routing_key))

        # NB: memory.write will automatically truncate any excess stimulus
        memory.write(bitarray(
            self.stimulus.ljust(self._simulator.length, "0"),
            endian="little").tobytes())
```

Next, the routing tables and SpiNNaker applications are loaded using *load_routing_tables()* and *load_application()*:

```
# Load all routing tables
mc.load_routing_tables(routing_tables)

# Load all SpiNNaker application kernels
mc.load_application(application_map)
```

We now wait for the applications to reach their initial barrier, send the 'sync0' signal to start simulation and, finally, wait for the cores to exit.

```
# Wait for all six cores to reach the 'sync0' barrier
mc.wait_for_cores_to_reach_state("sync0", len(self._components))

# Send the 'sync0' signal to start execution and wait for the
# simulation to finish.
mc.send_signal("sync0")
time.sleep(self.length * 0.001)
mc.wait_for_cores_to_reach_state("exit", len(self._components))
```

The last step is to read back results from the machine. As with loading, we add a `_read_results` method to each component type which we call with a *MemoryIO* object from which it should read any results it requires:

```
for component, memory in memory_allocations.items():
    component._read_results(memory)
```

The `_read_results` method is a no-op for all but the `Probe` component whose implementation is as follows:

```
class Probe(object):
    def _read_results(self, memory):
        """Read back the probed results.

        Returns
        -------
        str
            A string of "0"s and "1"s, one for each millisecond of simulation.
        """
        # Seek to the simulation data and read it all back
        memory.seek(8)
        bits = bitarray(endian="little")
        bits.frombytes(memory.read())
        self.recorded_data = bits.to01()
```

Trying it out

Congratulations! Our circuit simulator is now complete! We can now run the example script we used to define our simulator's API and within a second or so we have our results!

```
$ python example_circuit.py HOSTNAME_OR_IP
Stimulus A: 0000000011111111000000001111111100000000111111110000000011111111
Stimulus B: 0000000000000000111111111111111100000000000000001111111111111111
Stimulus C: 0000000000000000000000000000000011111111111111111111111111111111
Probe:      0000000000000000000000000000000010000000111111111111111111111111
```

1.3 Tutorial: Controlling SpiNNaker machines

SpiNNaker machines consist of a network of SpiNNaker chips and, in larger systems, a set of Board Management Processors (BMPs) which control and monitor systems' power and temperature. SpiNNaker (and BMPs) are controlled using SCP packets (a protocol built on top of SDP) sent over the network to a machine. Rig includes a set of high-level wrappers around the low-level SCP commands which are tailored towards SpiNNaker application developers.

Note: Rig does not aim to provide a complete Python implementation of the full (low-level) SCP command set. Users who encounter missing functionality as a result of this are encouraged to submit a patch or open an issue as the developers are open to (reasonable) suggestions!

In addition to these high-level interfaces, Rig includes a lower-level interface for sending and receiving application-defined SDP and SCP packets to running applications via a socket.

The two high-level machine control interfaces are:

MachineController Interact with and control SpiNNaker chips, e.g. boot, load applications, read/write memory.

BMPController Interact with and control BMPs, e.g. control power-supplies, monitor system temperature, read/write FPGA registers. Only applicable to machines based on SpiNN-5 boards.

The low-level SDP and SCP interfaces are:

SDPPacket Pack and unpack SDP packets.

SCPPacket Pack and unpack SCP packets.

A tutorial for each of these interfaces is presented below.

1.3.1 MachineController

To get started, let's instantiate a *MachineController*. This is as simple as giving the hostname or IP address of the machine:

```
>>> from rig.machine_control import MachineController
>>> mc = MachineController("spinnaker_hostname")
```

Note: If you're using a multi-board machine, give the hostname of the (0, 0) chip. Support for connecting to multiple Ethernet ports of a SpiNNaker machine is not currently available but should be automatic.

Booting

You can *boot()* the system like so:

```
>>> mc.boot()
True
```

If the machine could not be booted for any reason a *rig.machine_control.machine_controller.SpiNNakerBootError* will be raised. If no exception is raised, the machine is booted and ready to use. The return value of *boot()* indicates whether the machine was actually booted (`True`), or if it was already booted and thus nothing was done (`False`), most applications may consider the boot to be a success either way.

If you're using a SpiNN-2 or SpiNN-3 board booted without arguments, only LED 0 will be usable. To enable the other LEDs, instead boot the machine using one of the pre-defined boot option dictionaries in *rig.machine_control.boot*, for example:

```
>>> from rig.machine_control.boot import spin3_boot_options
>>> mc.boot(**spin3_boot_options)
True
```

Probing for Available Resources

The *get_system_info()* method returns a *SystemInfo* object describing which chips, links and cores are alive and also the SDRAM available:

```
>>> system_info = mc.get_system_info()
```

This object can also be used to guide Rig's place and route utilities (see *rig.place_and_route.place_and_route_wrapper*, *rig.place_and_route.utils.build_machine* and *rig.place_and_route.utils.build_core_constraints*).

Loading Applications

The *load_application()* method will, unsurprisingly, load an application onto an arbitrary set of SpiNNaker cores. For example, the following code loads the specified APLX file to cores 1, 2 and 3 of chip (0, 0) and cores 10 and 11 of chip (0, 1):

```
>>> targets = {(0, 0): set([1, 2, 3]),
...            (0, 1): set([10, 11])}
>>> mc.load_application("/path/to/app.aplx", targets)
```

Alternatively, this method accepts dictionaries mapping applications to targets, such as those produced by *rig.place_and_route.place_and_route_wrapper*.

load_application() verifies that all applications have been successfully loaded (re-attempting a small number of times if necessary). If not all applications could be loaded, a *SpiNNakerLoadingError* exception is raised.

Many applications require the *sync0* signal to be sent to start the application's event handler after loading. We can wait for all cores to reach the *sync0* barrier using `wait_for_cores_to_reach_state` and then send the *sync0* signal using `send_signal`:

```
>>> # In the example above we loaded 5 cores so we expect 5 cores to reach
>>> # sync0.
>>> mc.wait_for_cores_to_reach_state("sync0", 5)
5
>>> mc.send_signal("sync0")
```

Similarly, after application execution, the application can be killed with:

```
>>> mc.send_signal("stop")
```

Since the stop signal also cleans up allocated resources in a SpiNNaker machine (e.g. stray processes, routing entries and allocated SDRAM), it is desirable for this signal to reliably get sent even if something crashes in the host application. To facilitate this, you can use the *application()* context manager:

```
>>> with mc.application():
...     # Main application code goes here, e.g. loading applications,
...     # routing tables and SDRAM.
>>> # When the above block exits (even if due to an exception), the stop
>>> # signal will be sent to the application.
```

Note: Many application-oriented methods accept an *app_id* argument which is given a sensible default value. If the *MachineController.application()* context manager is given an app ID as its argument, this app ID will become the default *app_id* within the *with* block. See the section on context managers below for more details.

Loading Routing Tables

Routing table entries can be loaded using *load_routing_tables()* like so:

```
>>> routing_tables = {
...     (0, 0): [RoutingTableEntry(...), ...],
...     (0, 1): [RoutingTableEntry(...), ...],
...     ...
... }
>>> mc.load_routing_tables(routing_tables)
```

This command allocates and then loads the requested routing table entries onto each of the supplied chips. The supplied data structure matches that produced by *rig.place_and_route.place_and_route_wrapper()*.

Allocating/Writing/Reading SDRAM

Many SpiNNaker applications require the writing and reading of large blocks of SDRAM data. The recommended way of doing this is to allocate blocks of SDRAM using *sdram_alloc()* with an identifying 'tag'. The The SpiNNaker application can later use this tag number to look up the address of the allocated block of SDRAM. Not only does this

avoid the need to explicitly communicate SDRAM locations to the application it also allows SARK to safely allocate memory in the SDRAM.

read() and *write()* methods are provided which can read and write arbitrarily large blocks of data to and from memory in SpiNNaker:

```
>>> # Allocate 1024 bytes of SDRAM with tag '3' on chip (0, 0)
>>> block_addr = mc.sdram_alloc(1024, 3, 0, 0)
>>> mc.write(block_addr, b"Hello, world!")
>>> mc.read(block_addr, 13)
b"Hello, world!"
```

Rig also provides a file-like I/O wrapper (*MemoryIO*) which may prove easier to integrate into applications and also ensures reads and writes are constrained to the allocated region.

```
>>> # Allocate 1024 bytes of SDRAM with tag '3' on chip (0, 0)
>>> block = mc.sdram_alloc_as_filelike(1024, 3, 0, 0)
>>> block.write(b"Hello, world!")
>>> block.seek(0)
>>> block.read(13)
b"Hello, world!"
```

File-like views of memory can also be sliced to allow a single allocation to be safely divided between different parts of the application:

```
>>> hello = block[0:5]
>>> hello.read()
b"Hello"
```

The *sdram_alloc_for_vertices()* utility function is provided to allocate multiple SDRAM blocks simultaneously. This will be especially useful if you're using Rig's *place and route tools*, since the utility accepts the place-and-route tools' output format. For example:

```
>>> placements, allocations, application_map, routing_tables = \
...     rig.place_and_route.wrapper(...)
>>> from rig.machine_control.utils import sdram_alloc_for_vertices
>>> vertex_memory = sdram_alloc_for_vertices(mc, placements, allocations)

>>> # The returned dictionary maps from vertex to file-like wrappers
>>> vertex_memory[vertex].write(b"Hello, world!")
```

Context Managers

Many methods of *MachineController* require arguments such as *x*, *y*, *p* or *app_id* which can quickly lead to repetitive and messy code. To reduce the repetition Python's `with` statement can be used:

```
>>> # Within the block, all commands will affect chip (1, 2)
>>> with mc(x = 1, y = 2):
...     block_addr = mc.sdram_alloc(1024, 3)
...     mc.write(block_addr, b"Hello, world!")
```

1.3.2 BMPController

A limited set of utilities are provided for interacting with SpiNNaker BMPs which are contained in the *BMPController* class. In systems with either a single SpiNN-5 board or a single frame of SpiNN-5 boards which

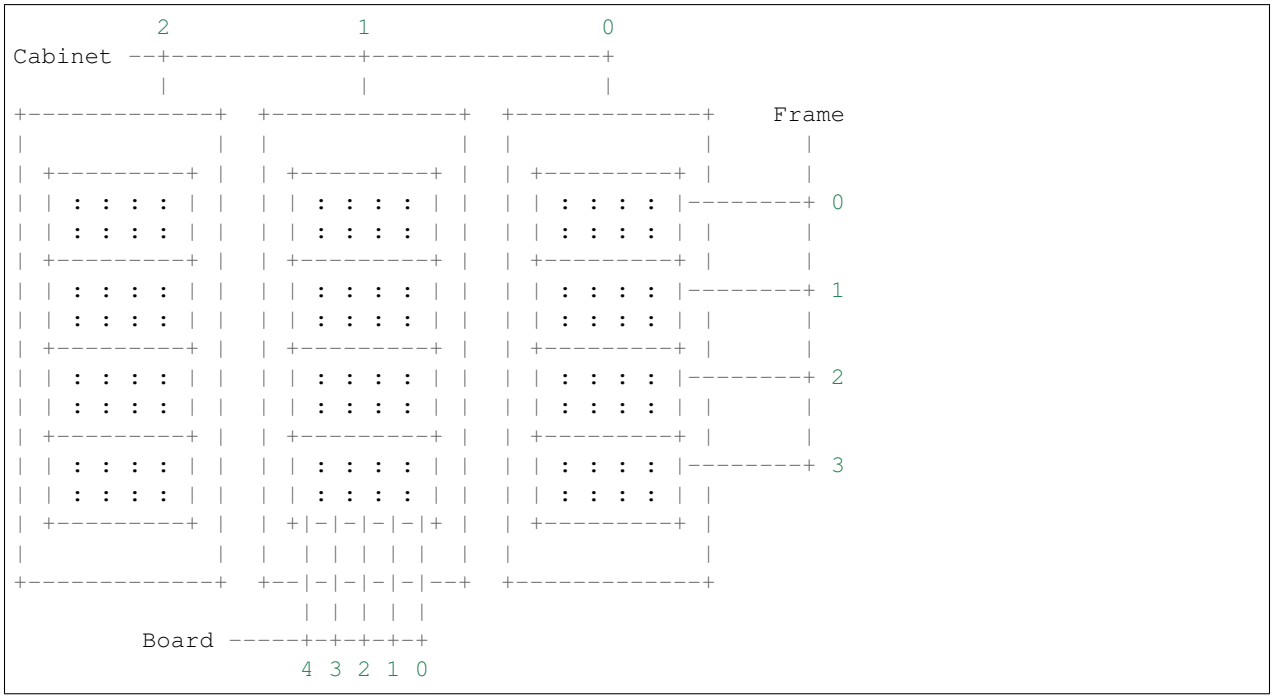
are connected via a backplane, the class can be constructed like so:

```
>>> from rig.machine_control import BMPController
>>> bc = BMPController("bmp_hostname")
```

For larger systems which contain many frames of SpiNNaker boards, at least one IP address or hostname must be specified for each:

```
>>> bc = BMPController({
...     # At least one hostname per rack is required
...     (0, 0): "cabinet0_frame0_hostname",
...     (0, 1): "cabinet0_frame1_hostname",
...     ...
...     (1, 0): "cabinet1_frame0_hostname",
...     (1, 1): "cabinet1_frame1_hostname",
...     ...
...     # Individual boards can be given their own unique hostname if
...     # required which overrides those above
...     (1, 1, 0): "cabinet1_frame1_board0_hostname",
... })
```

Boards are referred to by their (cabinet, frame, board) coordinates:



Power Control

Boards can be powered on using *set_power()*:

```
>>> # Power off board (0, 0, 0)
>>> bc.set_power(False)

>>> # Power on board (1, 2, 3)
>>> bc.set_power(True, 1, 2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> # Power on all 24 boards in frame (1, 2)
>>> bc.set_power(True, 1, 2, range(24))
```

Note: Though multiple boards in a single frame can be powered on simultaneously, boards in different frames must be powered on separately.

Note: By default the *set_power()* method adds a delay after the power on command has completed to allow time for the SpiNNaker cores to complete their self tests. If powering on many frames of boards, the *post_power_on_delay* argument can be used to reduce or eliminate this delay.

Reading Board Temperatures

Various information about a board's temperature and power supplies can be read using *read_adc()* (ADC = Analogue-to-Digital Converter) which returns a *bmp_controller.ADCInfo* named tuple containing many useful values:

```
>>> adc_info = bc.read_adc()  # Get info for board (0, 0, 0)
>>> adc_info.temp_top  # Celsius
23.125
>>> adc_info.fan_0  # RPM (or None if not attached)
2401
```

Context Managers

As with *MachineController*, *BMPController* supports the `with` syntax for specifying common arguments to a series of commands:

```
>>> with bc(cabinet=1, frame=2, board=3):
...     if bc.read_adc().temp_top > 75.0:
...         bc.set_led(7, True)  # Turn on LED 7 on the board
```

1.3.3 Sending/receiving SDP and SCP packets to/from applications

A number of low-level facilities are provided for users who wish to send and receive SCP and SDP packets directly. The most common use for these APIs is to send and receive SDP packets to and from a running SpiNNaker application to allow realtime monitoring and communication with the underlying application via an IP Tag. A minimal example of each is presented below.

Example: Sending SDP packets to a running application

In your SpiNNaker application you should register a callback handler for the arrival of SDP packets. For example, using the `spin1_api`:

```
spin1_callback_on(SDP_PACKET_RX, on_sdp_from_host, 0);
```

To send SDP packets to this application, you must open a UDP socket with which to send SDP packets to your SpiNNaker system. Note that (slightly confusingly) SpiNNaker listens for incoming SDP packets on the *SCP port*.

```
>>> import socket
>>> from rig.machine_control.consts import SCP_PORT
>>> out_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> out_sock.connect((hostname, SCP_PORT))
```

With the port opened, you can use the *rig.machine_control.packets.SDPPacket* and *rig.machine_control.packets.SCPPacket* classes to pack your data into properly formatted SDP or SCP packets. Since `sark` and `spin1_api` (unfortunately) make packing/unpacking SDP packets rather clumsy it is common to use SCP packets.

Note: SCP packets are just SDP packets with some additional fields placed in the SDP data payload. When a port number other than 0 is used SCP packets are passed to the application like any other SDP packet

---

As an example, to send an SCP packet core 1 on chip (0, 0) with a `cmd_rc` of `123`:

```
>>> from rig.machine_control.packets import SCPPacket
>>> data = b"Hello world!\0"
>>> packet = SCPPacket(
...     dest_port=1,
...     dest_x=0, dest_y=0, dest_cpu=1,
...     cmd_rc=123
...     data=data
... )
>>> out_sock.send(packet.bytestring)
```

On the receiving core the `on_sdp_from_host` callback might then look like this:

```
void on_sdp_from_host(uint mailbox, uint port)
{
  sdp_msg_t *msg = (sdp_msg_t *)mailbox;
  if (msg->cmd_rc == 123)
  {
    io_printf(IO_BUF,
              "Got SCP packet from host with data: %s\n",
              msg->data);
  }
  spin1_msg_free(msg);
}
```

Note: SpiNNaker can only receive packets up to a certain size. This size can be determined using *MachineController*'s *scp_data_length()* property This property defines the maximum length of the data-field in an SCP packet sent to the machine.

Example: Receiving SDP packets from a running application

To receive SDP packets from an application there must first be an open socket ready to receive the packets. For example:

```
>>> import socket
>>> PORT = 50007
>>> in_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> in_sock.bind(("", PORT))
```

Next, you must set up an 'IP tag' on every Ethernet-connected SpiNNaker chip through which SDP packets may be sent back to the host which informs SpiNNaker of the IP address these packets should be sent to.

A list of the Ethernet-connected chips in a typical SpiNNaker machine can be produced using *rig.machine_control.MachineController.get_system_info* and an IP tag configured on each using *rig.machine_control.MachineController.iptag_set* like so:

```
>>> from rig.machine_control import MachineController

>>> # Get the IP and port of the socket we opened
>>> addr, port = in_sock.getsockname()

>>> # Set-up IP Tag 1 on each ethernet-connected chip to forward all SDP
>>> # packets to this socket.
>>> mc = MachineController("spinnaker-machine-hostname")
>>> si = mc.get_system_info()
>>> for (x, y), chip_ip in si.ethernet_connected_chips():
...     mc.iptag_set(1, addr, port, x, y)
```

You can now listen for incoming packets and unpack them using *rig.machine_control.packets.SDPPacket.from_bytestring()* and *rig.machine_control.packets.SCPPacket.from_bytestring()*. For example, to unpack SCP packets received from the machine:

```
>>> from rig.machine_control.packets import SCPPacket
>>> while True:
...     data = self.in_sock.recv(512)
...     if not data:
...         break
...     packet = SCPPacket.from_bytestring(data)
...     print("Got SCP packet from core {packet.src_cpu} "
...           "of chip ({packet.src_x}, {packet.src_y}) "
...           "with cmd_rc {packet.cmd_rc} and data "
...           "{packet.data}.".format(packet=packet))
```

Note: We use a 512 byte UDP receive buffer since at present the largest SDP packet supported by the machine at the time of writing is 256 bytes + 24 bytes SCP header. Using power-of-two sized receive buffers is recommended on most operating systems for performance reasons. The *MachineController*'s *scp_data_length()* property can be used to get the actual value.

SCP packets might be sent from a SpiNNaker application using code such as:

```
sdp_msg_t msg;

void send_scp_packet(const char *data)
{
  // Send to the nearest Ethernet-connected chip.
  msg.tag = 1;
  msg.dest_port = PORT_ETH;
  msg.dest_addr = sv->eth_addr;
```

(continues on next page)

(continued from previous page)

```
  // Indicate the packet's origin as this chip/core. Note that the core is
  // indicated in the bottom 5 bits of the srce_port field.
  msg.flags = 0x07;
  msg.srce_port = spin1_get_core_id();
  msg.srce_addr = spin1_get_chip_id();

  // Copy the supplied data into the data field of the packet and update
  // the length accordingly.
  int len = strlen(data) + 1;  // Include the null-terminating byte
  spin1_memcpy(msg.data, (void *)data, len);
  msg.length = sizeof (sdp_hdr_t) + sizeof (cmd_hdr_t) + len;

  // and send it with a 100ms timeout
  spin1_send_sdp_msg(&msg, 100);
}
```

1.4 Tutorial: Selecting routing keys with `BitField`

In this tutorial we will tackle the commonly faced challenge of defining the SpiNNaker routing keys. In SpiNNaker, routing keys are 32-bit values which are used to uniquely identify multicast streams of packets flowing from one core to many others. We'll walk through a few simple example scenarios and demonstrate the key features of *BitField*s.

1.4.1 Defining a basic bit field

We'll start by defining a 32-bit bit field:

```
>>> from rig.bitfield import BitField
>>> b = BitField(32)
>>> b
<32-bit BitField>
```

Initially no fields are defined and so we must define some. Lets define the following fields:

chip Bits 31-16: The unique chip ID number of the chip which produced the packet.

core Bits 12-8: The core ID number of the core which produced the packet.

type Bits 7-0: Some application specific message-type indicator.

These fields can be defined like so:

```
>>> b.add_field("chip", length=16, start_at=16)
>>> b.add_field("core", length=5, start_at=8)
>>> b.add_field("type", length=8, start_at=0)
>>> b
<32-bit BitField 'chip':?, 'core':?, 'type':?>
```

We can now specify the value of these fields to define a specific routing key:

```
>>> TYPE_START = 0x01
>>> TYPE_STOP = 0x02
>>> # ...

>>> start_master = b(chip=1024, core=1, type=TYPE_START)
```

(continues on next page)

(continued from previous page)

```
>>> start_master
<32-bit BitField 'chip':1024, 'core':1, 'type':1>
```

Notice that a new *BitField* is produced but this one has its fields allocated specific values.

Note: The newly created *BitField* is linked to the original *BitField*. Amongst other things this means that if new fields are added to the original, they will also appear in this bit field. The utility of this will become more apparent later.

Since all the fields (and their lengths and positions) in *start_master* have been defined, we can use the *get_value()* and *get_mask()* methods to get the actual binary value of the bit field and also a mask which selects only those bits used by a field in the bit field:

```
>>> # Get the binary value of the bit field with these field values
>>> hex(start_master.get_value())
'0x4000101'

>>> # Get a mask which includes only fields in the bit field. (Note that this
>>> # bit field has a few bits in the middle which aren't part of any fields).
>>> hex(start_master.get_mask())
'0xffff1fff'
```

We don't have to define all the fields at once, however. We can also specify just some fields at a time like so:

```
>>> master_core = b(chip=1024, core=1)
>>> master_core
<32-bit BitField 'chip':1024, 'core':1, 'type':?>
```

This is useful because we can pass the *master_core* *BitField* around where fields are completed later:

```
>>> start_master = master_core(type=TYPE_START)
>>> stop_master = master_core(type=TYPE_STOP)

>>> start_master
<32-bit BitField 'chip':1024, 'core':1, 'type':1>
>>> hex(start_master.get_value())
'0x4000101'

>>> stop_master
<32-bit BitField 'chip':1024, 'core':1, 'type':2>
>>> hex(stop_master.get_value())
'0x4000102'
```

1.4.2 Automatically allocating fields to bits

In many cases, we don't really care exactly how our bit field is formatted. All we care is that the fields do not overlap and that they are large enough to represent the largest value assigned to that field. As a result, we can omit one or both of the *length* and *start_at* options to let *BitField* automatically allocate and position fields:

```
>>> # Starting a new example 32-bit bit field
>>> b = BitField(32)
>>> b.add_field("chip")
>>> b.add_field("core")
```

(continues on next page)

(continued from previous page)

```
>>> b.add_field("type")
>>> b
<32-bit BitField 'chip':?, 'core':?, 'type':?>
```

Note: It is perfectly valid to mix fields both with and without allocated lengths and positions. *BitField* will automatically verify that the fields created do not overlap.

Just as before, we can assign new values to each field:

```
>>> TYPE_START = 0x01
>>> TYPE_STOP = 0x02
>>> # ...

>>> start_master = b(chip=1024, core=1, type=TYPE_START)
>>> start_master
<32-bit BitField 'chip':1024, 'core':1, 'type':1>

>>> master_core = b(chip=1024, core=1)
>>> master_core
<32-bit BitField 'chip':1024, 'core':1, 'type':?>
>>> start_master = master_core(type=TYPE_START)
>>> start_master
<32-bit BitField 'chip':1024, 'core':1, 'type':1>
>>> stop_master = master_core(type=TYPE_STOP)
>>> stop_master
<32-bit BitField 'chip':1024, 'core':1, 'type':2>
```

At the moment, the three fields do not have a designated length or position in the bit field. Before we can use *get_value()* and *get_mask()* we must assign all fields a length and position using *assign_fields()*:

```
>>> # Oops: Fields haven't been assigned lengths and positions yet!
>>> hex(start_master.get_value())
Traceback (most recent call last):
ValueError: Field 'chip' does not have a fixed size/position.

>>> b.assign_fields()
>>> hex(start_master.get_value())
'0x1c00'
>>> hex(stop_master.get_value())
'0x2c00'
```

We can use *get_mask()* to see what bits in the bit field were allocated to each field like so:

```
>>> # What is the total set of bits used
>>> hex(b.get_mask())
'0x3fff'

>>> # Which bits are used for each field
>>> hex(b.get_mask(field="chip"))
'0x7ff'
>>> hex(b.get_mask(field="core"))
'0x800'
>>> hex(b.get_mask(field="type"))
'0x3000'
```

You'll see that the three fields have been assigned to three non-overlapping sets of bits in the bit field. We can also see that the *chip* field has been allocated 10 bits which is large enough to fit the largest value we assigned to that field, *1024*. Likewise, the *core* and *type* fields have been allocated one and two bits respectively to accommodate the values we provided.

Warning: When using dynamically lengthed/positioned fields, it is important that all bit field values are assigned before calling `assign_fields()`. If this is not the case, the fields may not be allocated adequate lengths to fit the values required. The implication of this is that applications should generally operate in two phases:

1. Assignment of field values (prior to `assign_fields()`)
2. Generation of binary values and masks (after `assign_fields()`)

1.4.3 Defining hierarchical bit fields

Fields can also exist in a hierarchical structure. For example, packets to/from external devices may use a different set of fields to those used internally between cores. In our example, we'll define that bit 31 of the key is *0* for internal packets and *1* for external packets. We can define this as a field as usual:

```
>>> # Starting another new example...
>>> b = BitField(32)
>>> b.add_field("external", length=1, start_at=31)
>>> b
<32-bit BitField 'external':?>
```

In this example, internal packets will have fields *chip*, *core* and *type* as before while external packets will have the fields *device_id* and *command*. These can be defined like so:

```
>>> # Internal fields
>>> b_internal = b(external=0)
>>> b_internal.add_field("chip")
>>> b_internal.add_field("core")
>>> b_internal.add_field("type")

>>> # External fields
>>> b_external = b(external=1)
>>> b_external.add_field("device_id")
>>> b_external.add_field("command")
```

Notice that to add fields which appear only when *external* is *0* or *1* we add them to the `BitField` with the *external* field set to the appropriate value.

Note: As mentioned earlier, all `BitField`s associated with the same bit field are linked and so adding fields to these derived `BitField` objects (i.e. *b_internal* and *b_external*) effects the whole bit field.

Now, whenever the *external* field is '0' we have fields *external*, *chip*, *core* and *type*. Whenever the *external* field is '1' we have fields *external*, *device_id* and *command*:

```
>>> b
<32-bit BitField 'external':?>
>>> b(external=0)
<32-bit BitField 'external':0, 'chip':?, 'core':?, 'type':?>
>>> b(external=1)
<32-bit BitField 'external':1, 'device_id':?, 'command':?>
```

Finally, defining values works exactly as we've seen before:

```
>>> # Setting all fields at once
>>> example_internal = b(external=0, chip=0, core=1, type=TYPE_START)
>>> example_internal
<32-bit BitField 'external':0, 'chip':0, 'core':1, 'type':1>
>>> example_external = b(external=1, device_id=0xBEEF, command=0x0)
>>> example_external
<32-bit BitField 'external':1, 'device_id':48879, 'command':0>

>>> # Setting fields incrementally
>>> master_core = b(external=0, chip=1, core=1)
>>> master_core
<32-bit BitField 'external':0, 'chip':1, 'core':1, 'type':?>
>>> start_master = master_core(type=TYPE_START)
>>> start_master
<32-bit BitField 'external':0, 'chip':1, 'core':1, 'type':1>

>>> # Assign fields to bits to see where things ended up
>>> b.assign_fields()
>>> hex(b.get_mask())
'0x80000000'
>>> hex(b(external=0).get_mask())
'0x80000007'
>>> hex(b(external=1).get_mask())
'0x8001ffff'
```

Because the *device_id* and *command* fields and the *chip*, *core* and *type* fields are never present in the same key, they may be allocated overlapping sets of bits. In this example, the lower bits of the bit field are used by both groups of fields depending on the value of *external*.

1.4.4 Selecting subsets of fields using tags

In many applications using bit fields, some fields are not relevant in every circumstance. For example, given our SpiNNaker routing key example, only the *chip* and *core* fields may be relevant to routing since the *type* field is only relevant to the receiving cores. As a result when building routing tables it is useful to only consider *chip* and *core* while in our application code we may only consider the *type*.

To facilitate this, fields can be labelled with tags like so:

```
>>> # Starting yet another new example...
>>> b = BitField(32)
>>> b.add_field("chip", tags="routing")
>>> b.add_field("core", tags="routing")
>>> b.add_field("type", tags="application")
>>> b
<32-bit BitField 'chip':?, 'core':?, 'type':?>
```

We can now use the *tag* arguments to *get_value()* and *get_mask()* to generate binary values and masks for just the fields with that tag:

```
>>> # Assign values like usual...
>>> master_core = b(chip=1024, core=1)
>>> stop_master = master_core(type=TYPE_STOP)
>>> b.assign_fields()
```

(continues on next page)

(continued from previous page)

```
>>> hex(master_core.get_value(tag="routing"))
'0xc00'
>>> hex(master_core.get_mask(tag="routing"))
'0xfff'
>>> hex(stop_master.get_value(tag="application"))
'0x2000'
>>> hex(stop_master.get_mask(tag="application"))
'0x3000'
```

Note: When used with a tag, *get_value()* only requires that the fields with the specified tag have a value. Notice how it could be successfully called on *master_core* with the tag *routing* which doesn't have the *type* field set.

When using hierarchical bit fields, assigning a tag to a field also assigns that tag to all fields above it in the hierarchy. For example in:

```
>>> # Starting yet another new example...
>>> b = BitField(32)
>>> b.add_field("external")

>>> b_internal = b(external=0)
>>> b_internal.add_field("chip", tags="routing")
>>> b_internal.add_field("core", tags="routing")
>>> b_internal.add_field("type", tags="application")

>>> b_external = b(external=1)
>>> b_external.add_field("device_id", tags="routing")
>>> b_external.add_field("command")
```

The following tags are assigned:

Field	Tags
external	routing, application
chip	routing
core	routing
type	application
device_id	routing
command	

This behaviour is important since fields with a given tag only exist when those further up the hierarchy have specific values. In other words: when checking that a given set of tagged fields have a certain value, we must equally check that those fields are present.

You can list the set of tags associated with a particular field using *get_tags()* like so:

```
>>> b_external.get_tags("device_id") == {'routing'}
True
```

1.4.5 Allowing 3rd party expansion of bit fields

In certain applications, it can be useful to allow two completely separate code-bases share the same bit field. For example, a SpiNNaker application may wish to support a range of plugins and as a result the application and its

plugins must be careful not to produce routing keys that interfere. Using the *BitField* class, it is possible to support this safely and simply like so:

```
>>> # Starting yet another additional new example...
>>> b = BitField(32)
>>> b.add_field("user")
>>> app_bitfield = b(user=0)
>>> plugin_1_bitfield = b(user=1)
>>> plugin_2_bitfield = b(user=2)
>>> plugin_3_bitfield = b(user=3)
>>> # ...
```

Each part of the application is then issued with its own *BitField* instance (e.g. *app_bitfield*, *plugin_1_bitfield* etc.) to which new fields may be assigned independently. These separate cases will never suffer any collisions since each user's bit fields are distinguished by the *user* field.

Note that field names need not be unique as long as fields which share the same name are never present at the same time. For example we can define:

```
>>> app_bitfield.add_field("command", length=8, start_at=0)
>>> plugin_1_bitfield.add_field("command", length=2, start_at=4)
```

In the above example, two completely independent fields, both named 'command', are created which exist only when `user=0` and `user=1` respectively. This has two useful side-effects:

- Users need not worry about field name collisions with fields defined by plugins.
- A common set of fields can be defined with different bit field layouts depending on the value of a particular field.

To more clearly demonstrate the utility of this, consider the (slightly contrived) case where we have two silicon retina devices, retina A and retina B with slightly different key formats. Retina A generates packets in response to light level changes whose key has the X coordinate of the pixel which changed in the bottom 8 bits of the key, and the Y coordinate in the next 8 bits. Retina B, however, has these two fields in the opposite order (Y is in the bottom 8 bits and X in the next 8).

```
>>> # One final example...
>>> b = BitField(32)
>>> RETINA_A = 1
>>> RETINA_B = 2
>>> b.add_field("retina", length=4, start_at=28)
>>> b_ra = b(retina=RETINA_A)
>>> b_rb = b(retina=RETINA_B)

>>> # We can define "x" and "y" fields for each retina
>>> b_ra.add_field("x", length=8, start_at=0)
>>> b_ra.add_field("y", length=8, start_at=8)

>>> b_rb.add_field("x", length=8, start_at=8)
>>> b_rb.add_field("y", length=8, start_at=0)

>>> # We can then generate keys the same way with either BitField
>>> for b_r in [b_ra, b_rb]:
...     print(hex(b_r(x=0x11, y=0x22).get_value()))
0x10002211
0x20001122
```

Warning: Though field names are namespaced as shown above making reusing field names safe and unambiguous (when allowed), tags are not. *This is a feature* since it allows the behaviours outlined in the section above on tags.

1.5 Quick-start examples in under 10 lines of code

This series of very short scripts and code snippets aim to give a concise demonstration of the key features of Rig in less than 10 'lines' (really statements) of Python.

1.5.1 Booting a SpiNNaker machine

```
>>> from rig.machine_control import MachineController

>>> mc = MachineController("hostname-or-ip")
>>> mc.boot()
True
```

Reference:

- *rig.machine_control.MachineController.boot()*

Tutorial:

- *MachineController tutorial*

1.5.2 Loading a SpiNNaker application

```
>>> from rig.machine_control import MachineController

>>> mc = MachineController("hostname-or-ip")

>>> # Load "app.aplx" onto cores 1, 2 and 3 of chip (0, 0) and cores 10 and
>>> # 11 of chip (0, 1).
>>> targets = {(0, 0): set([1, 2, 3]),
...            (0, 1): set([10, 11])}
>>> mc.load_application("app.aplx", targets)

>>> # Wait for the sync0 barrier, send the sync0 signal to start the
>>> # application, wait for it to exit
>>> mc.wait_for_cores_to_reach_state("sync0", 5)
5
>>> mc.send_signal("sync0")
>>> mc.wait_for_cores_to_reach_state("exit", 5)
5

>>> # Clean up!
>>> mc.send_signal("stop")
```

Reference:

- *rig.machine_control.MachineController.load_application()*
- *rig.machine_control.MachineController.wait_for_cores_to_reach_state()*
- *rig.machine_control.MachineController.send_signal()*

Tutorial:

- *MachineController tutorial*

1.5.3 Real-time communication via Ethernet using SDP

```
>>> import socket
>>> from rig.machine_control import MachineController
>>> from rig.machine_control.packets import SCPPacket

>>> # Open a UDP socket to receive packets on
>>> in_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> in_sock.bind(("", 50007))
>>> addr, port = in_sock.getsockname()

>>> # Set-up IP Tag 1 on chip (0, 0) to forward SDP packets the UDP socket
>>> mc = MachineController("spinnaker-machine-hostname")
>>> mc.iptag_set(1, addr, port, 0, 0)

>>> # Start receiving packets from an application running on SpiNNaker
>>> while True:
...     print(SCPPacket.from_bytestring(self.in_sock.recv(512)))
```

Reference:

- `socket`
- *`rig.machine_control.MachineController.iptag_set()`*
- *`rig.machine_control.packets.SCPPacket`*

Tutorial:

- *Sending/receiving SDP and SCP packets to/from applications*

1.5.4 Place and route

```
>>> import random
>>> from rig.place_and_route import place_and_route_wrapper, Cores, SDRAM
>>> from rig.netlist import Net
>>> from rig.machine_control import MachineController

>>> # Define a graph with 50 vertices and random 100 multicast nets.
>>> vertices = [object() for _ in range(50)]
>>> vertices_resources = {
...     vertex: {Cores: 1, SDRAM: 10 * 1024 * 1024}
...     for vertex in vertices
... }
>>> nets = [Net(random.choice(vertices), random.sample(vertices, 4))
...         for _ in range(100)]
>>> vertices_applications = {vertex: "my_app.aplx" for vertex in vertices}
>>> net_keys = {net: (number, 0xFFFFFFFF) for number, net in enumerate(nets)}

>>> # Interrogate the SpiNNaker machine to determine its topology etc.
>>> system_info = MachineController("hostname-or-ip").get_system_info()

>>> # Place, route and generate routing tables.
```

(continues on next page)

(continued from previous page)

```
>>> placements, allocations, application_map, routing_tables = \
...     place_and_route_wrapper(vertices_resources, vertices_applications,
...                             nets, net_keys, system_info)
```

Reference:

- *rig.place_and_route*
- *rig.place_and_route.place_and_route_wrapper()*
- *rig.machine_control.MachineController*

Tutorial:

- *05: Circuit Simulation*

1.5.5 Place and route for external devices

```
>>> # Assuming a graph defined as in the place-and-route example, lets add
>>> # a new vertex representing a device (e.g. as a silicon retina) directly
>>> # attached to the 'West' link of chip (0, 0), e.g. via a 2-of-7 or
>>> # S-ATA link.

>>> from rig.place_and_route import place_and_route_wrapper
>>> from rig.place_and_route.constraints import \
...     LocationConstraint, RouteEndpointConstraint
>>> from rig.routing_table import Routes
>>> from rig.netlist import Net

>>> # Make a vertex to represent the device which consumes no Cores or
>>> # SDRAM.
>>> device_vertex = object()
>>> vertices_resources[device_vertex] = {}

>>> # Use a pair of constraints to indicate that the vertex is attached
>>> # to the West link of (0, 0).
>>> constraints = [
...     LocationConstraint(device_vertex, (0, 0)),
...     RouteEndpointConstraint(device_vertex, Routes.west),
... ]

>>> # Any Net sourced or sunk by our device_vertex will be routed down the
>>> # appropriate link.

>>> # The constraint list must be passed in during place and route.
>>> placements, allocations, application_map, routing_tables = \
...     place_and_route_wrapper(vertices_resources, vertices_applications,
...                             nets, net_keys, system_info, constraints)
```

Reference:

- *rig.place_and_route*
- *rig.place_and_route.constraints*

1.5.6 Discover an unbooted SpiNNaker board's IP address

Via the command-line:

$ rig-discover 192.168.240.253

From Python:

```
>>> from rig.machine_control.unbooted_ping import listen
>>> listen()
"192.168.240.253"
```

Refrerence:

- *rig.machine_control.unbooted_ping.listen()*
- *rig-discover*

1.5.7 Discover if your application is dropping packets

$ rig-counters HOSTNAME –command python my_application.py HOSTNAME time,dropped_multicast 10.4,102

Our application took 10.4 seconds to execute and dropped 102 multicast packets in total.

Tutorial:

- *rig-counters*

1.5.8 Fixed point number conversion

```
>>> from rig.type_casts import float_to_fp, fp_to_float

>>> # Create a function to convert a float to a signed fractional
>>> # representation with 8 bits overall and 4 fractional bits (S3.4)
>>> s34 = float_to_fp(signed=True, n_bits=8, n_frac=4)
>>> hex(int(s34(0.5)))
'0x8'
>>> hex(int(s34(-7.5)))
'-0x78'

>>> # ...and make a function to convert back again!
>>> f4 = fp_to_float(n_frac=4)
>>> f4(0x08)
0.5
>>> f4(-0x78)
-7.5
```

Reference:

- *rig.type_casts.float_to_fp()*
- *rig.type_casts.fp_to_float()*

1.5.9 Fixed point number conversion (for Numpy)

```
>>> import numpy as np
>>> from rig.type_casts import \
...     NumpyFloatToFixConverter, NumpyFixToFloatConverter

>>> # Create a function to convert a float to a signed fractional
>>> # representation with 8 bits overall and 4 fractional bits (S3.4)
>>> s34 = NumpyFloatToFixConverter(signed=True, n_bits=8, n_frac=4)
>>> vals = np.array([0.0, 0.25, 0.5, -0.5, -0.25])
>>> s34(vals)
array([ 0,  4,  8, -8, -4], dtype=int8)

>>> # ...and make a function to convert back again!
>>> f4 = NumpyFixToFloatConverter(4)
>>> vals = np.array([ 0,  4,  8, -8, -4], dtype=np.int8)
>>> f4(vals)
array([ 0.  ,  0.25,  0.5 , -0.5 , -0.25])
```

Reference:

- *rig.type_casts.NumpyFloatToFixConverter*
- *rig.type_casts.NumpyFixToFloatConverter*

1.5.10 'XYP' routing keys with BitField

```
>>> from rig.bitfield import BitField

>>> # Define a classic 'XYP' routing key format
>>> b = BitField(32)
>>> b.add_field("x", length=8, start_at=24)
>>> b.add_field("y", length=8, start_at=16)
>>> b.add_field("p", length=5, start_at=11)
>>> b.add_field("neuron", length=11, start_at=0)

>>> # Define some keys
>>> my_favourite_core = b(x=1, y=2, p=3)
>>> neurons = [my_favourite_core(neuron=n) for n in range(10)]
>>> for neuron in neurons:
...     print(hex(neuron.get_value()))
0x1021800
0x1021801
0x1021802
0x1021803
0x1021804
0x1021805
0x1021806
0x1021807
0x1021808
0x1021809
```

Reference:

- *rig.bitfield.BitField*

Tutorial:

- *Tutorial: Selecting routing keys with BitField*

1.5.11 Hierarchical routing keys with BitField

```
>>> from rig.bitfield import BitField

>>> # Define two types of key, distinguished by bit 31
>>> b = BitField(32)
>>> b.add_field("type", length=1, start_at=31)
>>> type_0 = b(type=0)
>>> type_1 = b(type=1)

>>> # Each type can have different and overlapping fields
>>> type_0.add_field("magic", length=8, start_at=0)
>>> type_1.add_field("science", length=8, start_at=4)

>>> # Define some keys
>>> print(hex(type_0(magic=0xAB).get_value()))
0xab
>>> print(hex(type_1(science=0xCD).get_value()))
0x80000cd0

>>> # Can't access fields from other parts of the hierarchy
>>> type_0(science=123)
Traceback (most recent call last):
  ...
rig.bitfield.UnavailableFieldError: Field 'science' is not available when 'type':1.
```

Reference:

- *rig.bitfield.BitField*

Tutorial:

- *Tutorial: Selecting routing keys with BitField*

CHAPTER 2

Reference manual

The Rig reference manual describes Rig's public APIs, grouped by function. Most of this documentation is also accessible using Python's `help()` facility.

2.1 Data packaging for SpiNNaker

2.1.1 `rig.type_casts`: numerical type conversions for SpiNNaker

Fixed point conversion utilities.

`rig.type_casts.`**`float_to_fp`**(*signed*, *n_bits*, *n_frac*)

Return a function to convert a floating point value to a fixed point value.

For example, a function to convert a float to a signed fractional representation with 8 bits overall and 4 fractional bits (S3.4) can be constructed and used with:

```
>>> s34 = float_to_fp(signed=True, n_bits=8, n_frac=4)
>>> hex(int(s34(0.5)))
'0x8'
```

The fixed point conversion is saturating:

```
>>> q34 = float_to_fp(False, 8, 4)  # Unsigned 4.4
>>> hex(int(q34(-0.5)))
'0x0'

>>> hex(int(q34(15.0)))
'0xf0'

>>> hex(int(q34(16.0)))
'0xff'
```

Parameters

signed [bool] Whether the values that are to be converted should be signed, or clipped at zero.

```
>>> hex(int(float_to_fp(True, 8, 4)(-0.5)))  # Signed
'-0x8'
>>> hex(int(float_to_fp(False, 8, 4)(-0.5)))  # Unsigned
'0x0'
```

n_bits [int] Total number of bits in the fixed-point representation (including sign bit and fractional bits).

n_frac [int] Number of fractional bits in the fixed-point representation.

rig.type_casts.**fp_to_float**(*n_frac*)

Return a function to convert a fixed point value to a floating point value.

For example, a function to convert from signed fractional representations with 4 fractional bits constructed and used with:

```
>>> f = fp_to_float(4)
>>> f(0x08)
0.5

>>> f(-0x8)
-0.5

>>> f(-0x78)
-7.5
```

Parameters

n_frac [int] Number of fractional bits in the fixed-point representation.

rig.type_casts.**float_to_fix**(*signed*, *n_bits*, *n_frac*)

DEPRECATED Return a function to convert a floating point value to a fixed point value.

> **Warning:** This function is deprecated in favour of `float_to_fp()`.

For example, a function to convert a float to a signed fractional representation with 8 bits overall and 4 fractional bits (S3.4) can be constructed and used with:

```
>>> s34 = float_to_fix(signed=True, n_bits=8, n_frac=4)
>>> hex(s34(0.5))
'0x8'
```

The fixed point conversion is saturating:

```
>>> q34 = float_to_fix(False, 8, 4)  # Unsigned 4.4
>>> hex(q34(-0.5))
'0x0'

>>> hex(q34(15.0))
'0xf0'

>>> hex(q34(16.0))
'0xff'
```

Parameters

signed [bool] Whether the values that are to be converted should be signed, or clipped at zero.

```
>>> hex(float_to_fix(True, 8, 4)(-0.5))  # Signed
'0xf8'
>>> hex(float_to_fix(False, 8, 4)(-0.5))  # Unsigned
'0x0'
```

Note: Regardless of the value of the *signed* parameter the returned value is always an unsigned integer suitable for packing with the struct packing chars *B*, *H*, *I* etc.

n_bits [int] Total number of bits in the fixed-point representation (including sign bit and fractional bits).

n_frac [int] Number of fractional bits in the fixed-point representation.

Raises

ValueError If the number of bits specified is not possible. For example, requiring more fractional bits than there are bits overall will result in a *ValueError*:

```
>>> fix_to_float(False, 8, 9)
Traceback (most recent call last):
ValueError: n_frac: 9: Must be less than 8 (and positive).
```

rig.type_casts.**fix_to_float**(*signed*, *n_bits*, *n_frac*)

DEPRECATED Return a function to convert a fixed point value to a floating point value.

Warning: This function is deprecated in favour of `fp_to_float()`.

For example, a function to convert from signed fractional representations with 8 bits overall and 4 fractional representations (S3.4) can be constructed and used with:

```
>>> f = fix_to_float(True, 8, 4)
>>> f(0x08)
0.5

>>> f(0xf8)
-0.5

>>> f(0x88)
-7.5
```

Parameters

signed [bool] Determines whether input values should be treated as signed or otherwise, e.g.:

```
>>> fix_to_float(True, 8, 4)(0xfc)
-0.25

>>> fix_to_float(False, 8, 4)(0xf8)
15.5
```

The value accepted by the returned function should always be an unsigned integer.

n_bits [int] Total number of bits in the fixed-point representation (including sign bit and fractional bits).

n_frac [int] Number of fractional bits in the fixed-point representation.

Raises

ValueError If the number of bits specified is not possible. For example, requiring more fractional bits than there are bits overall will result in a *ValueError*:

```
>>> fix_to_float(False, 8, 9)
Traceback (most recent call last):
ValueError: n_frac: 9: Must be less than 8 (and positive).
```

class rig.type_casts.**NumpyFloatToFixConverter**(*signed*, *n_bits*, *n_frac*)

A callable which converts Numpy arrays of floats to fixed point arrays.

General usage is to create a new converter and then call this on arrays of values. The *dtype* of the returned array is determined from the parameters passed. For example:

```
>>> f = NumpyFloatToFixConverter(signed=True, n_bits=8, n_frac=4)
```

Will convert floating point values to 8-bit signed representations with 4 fractional bits. Consequently the returned *dtype* will be *int8*:

```
>>> import numpy as np
>>> vals = np.array([0.0, 0.25, 0.5, -0.5, -0.25])
>>> f(vals)
array([ 0,  4,  8, -8, -4], dtype=int8)
```

The conversion is saturating:

```
>>> f(np.array([15.0, 16.0, -16.0, -17.0]))
array([ 127,  127, -128, -128], dtype=int8)
```

The byte representation can be expected to match that for using *float_to_fix*:

```
>>> d = f(np.array([-16.0]))

>>> import struct
>>> g = float_to_fix(True, 8, 4)
>>> val = g(-16.0)
>>> struct.pack('B', val) == bytes(d.data)
True
```

An exception is raised if the number of bits specified cannot be represented using a whole *dtype*:

```
>>> NumpyFloatToFixConverter(True, 12, 0)
Traceback (most recent call last):
ValueError: n_bits: 12: Must be 8, 16, 32 or 64.
```

class rig.type_casts.**NumpyFixToFloatConverter**(*n_frac*)

A callable which converts Numpy arrays of fixed point values to floating point.

General usage is to create a new converter and then call this on arrays of values. The *dtype* of the input array is used to determine whether the values are signed or otherwise. For example, the following creates a callable which will convert from any format which has 4 fractional bits:

```
>>> kbits = NumpyFixToFloatConverter(4)
```

This will produced signed and unsigned values depending on the *dtype* of the original array.

```
>>> signed = np.array([0xf0], dtype=np.int8)
>>> kbits(signed)
array([-1.])
```

```
>>> unsigned = np.array([0xf0], dtype=np.uint8)
>>> kbits(unsigned)[0]
15.0
```

2.1.2 `rig.bitfield`: routing key construction made easy

In many applications there is a need to define bit fields, for example when defining SpiNNaker routing keys. Rig provides the class *rig.bitfield.BitField* which allows the definition of hierarchical bit fields. The full API documentation is provided below and a *tutorial is also available in the Rig documentation*.

A tutorial introduction to this class is provided below and is followed by the full API documentation.

class rig.bitfield.**BitField**(*length=32*, *_fields=None*, *_field_values=None*)

Defines a hierarchical bit field and the values of those fields.

Conceptually, a bit field is a sequence of bits which are logically broken up into individual fields which represent independent, unsigned integer values. For example, one could represent a pair of eight-bit values x and y as a sixteen-bit bit field where the upper eight bits are x and the lower eight bits are y. Bit fields are used when multiple pieces of information must be conveyed by a single binary value.

For example, one method of allocating SpiNNaker routing keys (which are 32-bit values) is to define each route a key as bit field with three fields. The fields x, y, and p can be used to represent the x- and y-chip-coordinate and processor id of a route's source.

A hierarchical bit field is a bit field with fields which only exist dependent on the values of other fields. For a further routing-key related example, different key formats may be used by external devices and the rest of the SpiNNaker application. In these cases, a single bit could be used in the key to determine which key format is in use. Depending on the value of this bit, different fields would become available.

This class supports the following key features:

- Construction of guaranteed-safe hierarchical bit field formats.
- Generation of bit-masks which select only defined fields
- Automatic allocation of field sizes based on values actually used.
- Partial-definition of a bit field (i.e. defining only a subset of available fields).

__init__(*length=32*, *_fields=None*, *_field_values=None*)

Create a new BitField.

An instance, *b*, of *BitField* represents a fixed-length hierarchical bit field with initially no fields. Fields can be added using *BitField.add_field()*. Derivatives of this instance with fields set to specific values can be created using the 'call' syntax: *b(field_name=value, other_field_name=other_value)* (see *BitField.__call__()*).

Note: Only one *BitField* instance should be explicitly created for each bit field.

Parameters

length [int] The total number of bits in the bit field.

_fields [`_Tree`] For internal use only. The shared, global field tree.

_field_values [dict] For internal use only. Mapping of field-identifier to value.

add_field(*identifier*, *length=None*, *start_at=None*, *tags=None*)

Add a new field to the BitField.

If any existing fields' values are set, the newly created field will become a child of those fields. This means that this field will exist only when the parent fields' values are set as they are currently.

Parameters

identifier [str] A identifier for the field. Must be a valid python identifier. Field names must be unique within the scope in which they exist and are only valid within that scope. For example:

```
>>> bf = BitField(32)
>>> bf.add_field("a")

>>> # Can add multiple fields with the same name if they exist
>>> # in different scopes
>>> bf0 = bf(a=0)
>>> bf0.add_field("b", length=4)
>>> bf1 = bf(a=1)
>>> bf1.add_field("b", length=8)

>>> # Can't add multiple fields with the same name which exist
>>> # within the same or nested scopes.
>>> bf.add_field("a")
Traceback (most recent call last):
ValueError: Field with identifier 'a' already exists
>>> bf.add_field("b")
Traceback (most recent call last):
ValueError: Field with identifier 'b' already exists
```

Here *three* fields are defined, one called "a" and the other two called "b". The two fields called "b" are completely unrelated (they may differ in size, position and associated set of tags) and are distinguished by the fact that one exists when a=0 and the other when a=1.

length [int or None] The number of bits in the field. If None the field will be automatically assigned a length long enough for the largest value assigned.

start_at [int or None] 0-based index of least significant bit of the field within the bit field. If None the field will be automatically located in free space in the bit field.

tags [string or collection of strings or None] A (possibly empty) set of tags used to classify the field. Tags should be valid Python identifiers. If a string, the string must be a single tag or a space-separated list of tags. If *None*, an empty set of tags is assumed. These tags are applied recursively to all fields of which this field is a child.

Raises

ValueError If any the field overlaps with another one or does not fit within the bit field. Note that fields with unspecified lengths and positions do not undergo such checks until their length and position become known when *`assign_fields()`* is called.

__call__(***field_values*)

Return a new BitField instance with fields assigned values as specified in the keyword arguments.

Returns

:py:class:‘.BitField‘ A *BitField* derived from this one but with the specified fields assigned a value.

Raises

ValueError If any field has already been assigned a value or the value is too large for the field.

UnavailableFieldError If a field is specified which does not exist or is not available.

__getattr__(*identifier*)
Get the value of a field.

Returns

int or None The value of the field (or None if the field has not been given a value).

Raises

UnavailableFieldError If the field requested does not exist or is not available given current field values.

get_value(*tag=None*, *field=None*)
Generate an integer whose bits are set according to the values of fields in this bit field. All other bits are set to zero.

Parameters

tag [str] Optionally specifies that the value should only include fields with the specified tag.

field [str] Optionally specifies that the value should only include the specified field.

Raises

ValueError If a field’s value, length or position has not been defined. (e.g. *assign_fields()* has not been called).

UnknownTagError If the tag specified using the *tag* argument does not exist.

UnavailableFieldError If the field specified using the *field* argument does not exist or is not available.

get_mask(*tag=None*, *field=None*)
Get the mask for all fields which exist in the current bit field.

Parameters

tag [str] Optionally specifies that the mask should only include fields with the specified tag.

field [str] Optionally specifies that the mask should only include the specified field.

Raises

ValueError If a field’s length or position has not been defined. (e.g. *assign_fields()* has not been called).

UnknownTagError If the tag specified using the *tag* argument does not exist.

UnavailableFieldError If the field specified using the *field* argument does not exist or is not available.

get_tags(*field*)
Get the set of tags for a given field.

Note: The named field must be accessible given the current set of values defined.

Parameters

field [str] The field whose tag should be read.

Returns

set([tag, ...])

Raises

UnavailableFieldError If the field does not exist or is not available.

get_location_and_length(*field*)
Get the location and length of a field within the bitfield.

Note: The named field must be accessible given the current set of values defined.

Parameters

field [str] The field of interest.

Returns

location, length A pair of integers defining the bit-number of the least-significant bit in the field and the total number of bits in the field respectively.

Raises

ValueError If a field's length or position has not been defined. (e.g. *assign_fields()* has not been called).

UnavailableFieldError If the field does not exist or is not available.

assign_fields()
Assign a position & length to any fields which do not have one.

Users should typically call this method after all field values have been assigned, otherwise fields may be fixed at an inadequate size.

__eq__(*other*)
Test that this *BitField* is equivalent to another.

In order to be equal, the other *BitField* must be a descendent of the same original *BitField* (and thus will *always* have exactly the same set of fields). It must also have the same field values defined.

__ne__(*other*)
x.__ne__(y) <==> x!=y

__repr__()
Produce a human-readable representation of this bit field and its current value.

__weakref__
list of weak references to the object (if defined)

class rig.bitfield.**UnknownTagError**(*tag*)
Exception thrown when a tag is specified which does not exist.

class rig.bitfield.**UnavailableFieldError**(*tree*, *identifier*, *field_values*)
Exception thrown when a field is requested from a BitField which is not does not exist or is unavailable (i.e. not in scope).

2.2 Graph-to-machine mapping

2.2.1 rig.place_and_route: Place applications, allocate on-chip resources, create routing tables

Rig provides a set of algorithms and utilities for mapping graph-like applications onto specific cores in a SpiNNaker machine and defining routes between them. Broadly, the task is split into three steps:

- **Placement**: Assign graph vertices to a chips.
- **Allocation**: Allocate specific chip resources to each vertex (e.g. cores, memory).
- **Routing**: Generate routes to connect vertices according to a supplied set of nets.

Rig provides a selection of complementary algorithm implementations for each step which attempt to carry out these tasks. Users are, of course, free to use their own application-specific processes in place of any or all of these steps.

Terminology

The key pieces of terminology used are defined below:

Application Graph The hyper-graph which describes how an application's computational resources (the *vertices*) are connected to each other by *nets*.

Vertex A *vertex* in an *application graph*. Each vertex is mapped onto exactly one SpiNNaker chip by during the placement process. (Note: an individual SpiNNaker chip may have several *vertices* mapped to it). A vertex may consume a certain set of *resources*. In most applications a vertex will represent an application to be run on a single SpiNNaker core.

Vertices are represented by application-defined `objects` which implement `object.__eq__()` and `object.__hash__()`.

Net A (directed) connection from one *vertex* to a number of other *vertices* in the *application graph*. During routing, nets are converted into specific routes through a SpiNNaker machine which can be used to generate routing tables.

Nets are represented by instances of the *rig.netlist.Net* class.

Resource A *resource* is any finite resource available to a SpiNNaker chip (e.g. SDRAM) which may be consumed by a vertex. *Resources* are allocated to each *vertex* during allocation. Users are welcome to define their own application-specific resources.

The type of a *resource* is represented by some unique Python `object`. Some common resources are defined in *rig.place_and_route* (though users are free to use their own):

- *rig.place_and_route.Cores*: Resource identifier for (monitor and application) processor cores.
- *rig.place_and_route.SDRAM*: Resource identifier for shared off-die SDRAM (in bytes).
- *rig.place_and_route.SRAM*: Resource identifier for shared on-die SRAM (in bytes).

Quantities of a *resource* are represented by positive integer values.

Constraint *Constraints* specify additional requirements on how an application graph is placed and routed. For example a constraint might be used to force a particular *vertex* to always be placed on a specific chip.

A number of types of *constraint* are defined in `rig.place_and_route.constraints`.

Note: It is worth emphasising that vertices are placed on SpiNNaker *chips*, not specific cores. In this library, cores are just one of many chip resources which vertices may consume.

For most applications, each vertex represents exactly one core worth of work and so each vertex will consume a single core of spinnaker chip resource.

Vertices which consume no cores are typically only useful when describing external devices connected to the SpiNNaker system.

Vertices which consume more than one core are unlikely to be used frequently:

- Vertices will always be placed on a single SpiNNaker chip: they cannot be split across many chips. If an application requires this type of behaviour, users must perform this step in an application-defined process prior to placement.
- If several cores' applications must share some on-chip resource (e.g. SDRAM) and must be placed on the same chip, a `SameChipConstraint` can be used to enforce this requirement. Unlike a vertex with multiple cores, each individual vertex (core) can have independent routes directly to and from them.

`place_and_route_wrapper()`: common case wrapper

Most applications simply require their application graph be translated into a set of data structures describing where binaries need to be loaded and a set of routing tables. For most users the `rig.place_and_route.place_and_route_wrapper()` will do exactly this with a minimum of fuss. For more advanced users, this function can be used as an example of the complete place-and-route process.

rig.place_and_route.**place_and_route_wrapper**(*vertices_resources*, *vertices_applications*, *nets*, *net_keys*, *system_info*, *constraints=[]*, *place=<function place>*, *place_kwargs={}*, *allocate=<function allocate>*, *allocate_kwargs={}*, *route=<function route>*, *route_kwargs={}*, *minimise_tables_methods=(<function minimise>*, *<function minimise>)*, *core_resource=Cores*, *sdram_resource=SDRAM*, *sram_resource=SRAM*)

Wrapper for core place-and-route tasks for the common case.

This function takes a set of vertices and nets and produces placements, allocations, minimised routing tables and application loading information.

Note: This function replaces the deprecated `wrapper()` function and makes use of the additional information provided by the `SystemInfo` object to infer the constraints required by most applications such as reserving non-idle cores such as the monitor processor.

Parameters

vertices_resources [{vertex: {resource: quantity, ...}, ...}] A dictionary from vertex to the required resources for that vertex. This dictionary must include an entry for every vertex in the application.

Resource requirements are specified by a dictionary *{resource: quantity, ...}* where *resource* is some resource identifier and *quantity* is a non-negative integer representing the quantity of that resource required.

vertices_applications [{vertex: application, ...}] A dictionary from vertices to the application binary to load onto cores associated with that vertex. Applications are given as a string containing the file name of the binary to load.

nets [[*Net*, ...]] A list (in no particular order) defining the nets connecting vertices.

net_keys [{*Net*: (key, mask), ...}] A dictionary from nets to (key, mask) tuples to be used in SpiNNaker routing tables for routes implementing this net. The key and mask should be given as 32-bit integers.

system_info [*SystemInfo*] A data structure which defines the resources available in the target SpiNNaker machine, typically returned by *rig.machine_control.MachineController.get_system_info()*. This information will be used internally to build a *Machine* and set of *rig.place_and_route.constraints* which describe the SpiNNaker machine used and ensure placement, allocation and routing only use working and unused chips, cores, memory and links. If greater control over these datastructures is required this wrapper may not be appropriate.

constraints [[constraint, ...]] **Optional.** A list of additional constraints on placement, allocation and routing. Available constraints are provided in the *rig.place_and_route.constraints* module. These constraints will be added to those derrived from the `system_info` argument which restrict placement and allocation to only idle cores.

place [function (Default: *rig.place_and_route.place()*)] **Optional.** Placement algorithm to use.

place_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the placer.

allocate [function (Default: *rig.place_and_route.allocate()*)] **Optional.** Allocation algorithm to use.

allocate_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the allocator.

route [function (Default: *rig.place_and_route.route()*)] **Optional.** Routing algorithm to use.

route_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the router.

minimise_tables_methods [[*rig.routing_table.minimise()*, ...]] **Optional.** An iterable of routing table minimisation algorithms to use when routing tables outgrow the space available. Each method is tried in the order presented and the first to meet the required target length for a given chip is used. Consequently less computationally costly algorithms should be nearer the start of the list. The default methods will try to remove default routes (*rig.routing_table.remove_default_routes.minimise()*) and then fall back on the ordered covering algorithm (*rig.routing_table.ordered_covering.minimise()*).

core_resource [resource (Default: *Cores*)] **Optional.** The resource identifier used for cores.

sdram_resource [resource (Default: *SDRAM*)] **Optional.** The resource identifier used for SDRAM.

sram_resource [resource (Default: *SRAM*)] **Optional.** The resource identifier used for SRAM (System RAM).

Returns

placements [{vertex: (x, y), ...}] A dictionary from vertices to the chip coordinate produced by placement.

allocations [{vertex: {resource: slice, ...}, ...}] A dictionary from vertices to the resources allocated to it. Resource allocations are dictionaries from resources to a `slice` defining the range of the given resource type allocated to the vertex. These `slice` objects have *start* <= *end* and *step* set to None.

application_map [{application: {(x, y): set([core_num, ...]), ...}, ...}] A dictionary from application to the set of cores it should be loaded onto. The set of cores is given as a dictionary from chip to sets of core numbers.

routing_tables [{(x, y): [*RoutingTableEntry*, ...], ...}] The generated routing tables. Provided as a dictionary from chip to a list of routing table entries.

rig.place_and_route.**wrapper**(*vertices_resources*, *vertices_applications*, *nets*, *net_keys*, *machine*, *constraints=[]*, *reserve_monitor=True*, *align_sdram=True*, *place=<function place>*, *place_kwargs={}*, *allocate=<function allocate>*, *allocate_kwargs={}*, *route=<function route>*, *route_kwargs={}*, *core_resource=Cores*, *sdram_resource=SDRAM*)

Wrapper for core place-and-route tasks for the common case. At a high level this function essentially takes a set of vertices and nets and produces placements, memory allocations, routing tables and application loading information.

> **Warning:** This function is deprecated. New users should use *place_and_route_wrapper()* along with *rig.machine_control.MachineController.get_system_info()* in place of this function. The new wrapper automatically reserves cores and SDRAM already in use in the target machine, improving on the behaviour of this wrapper which blindly reserves certain ranges of resources presuming only core 0 (the monitor processor) is not idle.

Parameters

vertices_resources [{vertex: {resource: quantity, ...}, ...}] A dictionary from vertex to the required resources for that vertex. This dictionary must include an entry for every vertex in the application. Resource requirements are specified by a dictionary *{resource: quantity, ...}* where *resource* is some resource identifier and *quantity* is a non-negative integer representing the quantity of that resource required.

vertices_applications [{vertex: application, ...}] A dictionary from vertices to the application binary to load onto cores associated with that vertex. Applications are given as a string containing the file name of the binary to load.

nets [[*Net*, ...]] A list (in no particular order) defining the nets connecting vertices.

net_keys [{*Net*: (key, mask), ...}] A dictionary from nets to (key, mask) tuples to be used in SpiNNaker routing tables for routes implementing this net. The key and mask should be given as 32-bit integers.

machine [*rig.place_and_route.Machine*] A data structure which defines the resources available in the target SpiNNaker machine.

constraints [[constraint, ...]] A list of constraints on placement, allocation and routing. Available constraints are provided in the *rig.place_and_route.constraints* module.

reserve_monitor [bool (Default: True)] **Optional.** If True, reserve core zero since it will be used as the monitor processor using a *rig.place_and_route.constraints.ReserveResourceConstraint*.

align_sdram [bool (Default: True)] **Optional.** If True, SDRAM allocations will be aligned to 4-byte addresses. Specifically, the supplied constraints will be augmented with an *AlignResourceConstraint(sdram_resource, 4)*.

place [function (Default: *rig.place_and_route.place()*)] **Optional.** Placement algorithm to use.

place_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the placer.

allocate [function (Default: *rig.place_and_route.allocate()*)] **Optional.** Allocation algorithm to use.

allocate_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the allocator.

route [function (Default: *rig.place_and_route.route()*)] **Optional.** Routing algorithm to use.

route_kwargs [dict (Default: {})] **Optional.** Algorithm-specific arguments for the router.

core_resource [resource (Default: *Cores*)] **Optional.** The resource identifier used for cores.

sdram_resource [resource (Default: *SDRAM*)] **Optional.** The resource identifier used for SDRAM.

Returns

placements [{vertex: (x, y), ...}] A dictionary from vertices to the chip coordinate produced by placement.

allocations [{vertex: {resource: slice, ...}, ...}] A dictionary from vertices to the resources allocated to it. Resource allocations are dictionaries from resources to a `slice` defining the range of the given resource type allocated to the vertex. These `slice` objects have *start* <= *end* and *step* set to None.

application_map [{application: {(x, y): set([core_num, ...]), ...}, ...}] A dictionary from application to the set of cores it should be loaded onto. The set of cores is given as a dictionary from chip to sets of core numbers.

routing_tables [{(x, y): [*RoutingTableEntry*, ...], ...}] The generated routing tables. Provided as a dictionary from chip to a list of routing table entries.

Placement, allocation and routing algorithms

The three key steps of the place-and-route process (placement, allocation and routing) are broken into three functions with a common API exposed by all algorithm implementations.

- *place()*
- *allocate()*
- *route()*

Since these tasks are largely NP-complete, rig attempts to include a selection of complimentary algorithms whose function prototypes are shared (and defined below) to allow users to easily swap between them as required.

Sensible default implementations for each function are aliased as:

- *rig.place_and_route.place()*
- *rig.place_and_route.allocate()*

- *rig.place_and_route.route()*

place() prototype

rig.place_and_route.**place**(*vertices_resources*, *nets*, *machine*, *constraints*, ***kwargs*)
Place vertices on specific chips.

The placement must be such that dead chips are not used and chip resources are not over-allocated.

vertices_resources [{vertex: {resource: quantity, ...}, ...}] A dictionary from vertex to the required resources for that vertex. This dictionary must include an entry for every vertex in the application.

Resource requirements are specified by a dictionary *{resource: quantity, ...}* where *resource* is some resource identifier and *quantity* is a non-negative integer representing the quantity of that resource required.

nets [[*Net*, ...]] A list (in no particular order) defining the nets connecting vertices.

machine [*rig.place_and_route.Machine*] A data structure which defines the resources available in the target SpiNNaker machine.

constraints [[constraint, ...]] A list of constraints on placement, allocation and routing. Available constraints are provided in the *rig.place_and_route.constraints* module.

****kwargs** Additional implementation-specific options.

{vertex: (x, y), ...} A dictionary from vertices to chip coordinate.

rig.place_and_route.exceptions.InvalidConstraintError If a constraint is impossible to meet.

rig.place_and_route.exceptions.InsufficientResourceError The placer could not find a placement where sufficient resources are available on each core.

allocate() prototype

rig.place_and_route.**allocate**(*vertices_resources*, *nets*, *machine*, *constraints*, *placements*, ***kwargs*)
Allocate chip resources to vertices.

vertices_resources [{vertex: {resource: quantity, ...}, ...}] A dictionary from vertex to the required resources for that vertex. This dictionary must include an entry for every vertex in the application.

Resource requirements are specified by a dictionary *{resource: quantity, ...}* where *resource* is some resource identifier and *quantity* is a non-negative integer representing the quantity of that resource required.

nets [[*Net*, ...]] A list (in no particular order) defining the nets connecting vertices.

machine [*rig.place_and_route.Machine*] A data structure which defines the resources available in the target SpiNNaker machine.

constraints [[constraint, ...]] A list of constraints on placement, allocation and routing. Available constraints are provided in the *rig.place_and_route.constraints* module.

placements [{vertex: (x, y), ...}] A dictionary of the format returned by *place()* describing a set of placements of vertices.

> **Warning:** The placement must not have vertices on dead/non-existent chips. failure to comply with this requirement will result in undefined behaviour.

****kwargs** Additional implementation-specific options.

{vertex: {resource: slice, ...}, ...} A dictionary from vertices to the resources allocated to it. Resource allocations are dictionaries from resources to a `slice` defining the range of the given resource type allocated to the vertex. These `slice` objects have *start* <= *end* and *step* set to None (i.e. resources are allocated to vertices in continuous blocks).

rig.place_and_route.exceptions.InvalidConstraintError If a constraint is impossible to meet.

rig.place_and_route.exceptions.InsufficientResourceError The allocator could not allocate all desired resources to those available.

`route()` prototype

`rig.place_and_route.`**`route`**(*vertices_resources*, *nets*, *machine*, *constraints*, *placements*, *allocations={}*, *core_resource=Cores*, ***kwargs*)

Generate routes which connect the vertices of all nets together.

This function produces a *`RoutingTree`* for each net which defines a multicast tree route through chips rooted at the chip containing the source vertex and visiting every chip on which a sink vertex is placed on. This data structure can then be converted into routing tables ready for loading onto a SpiNNaker machine using the *`rig.routing_table.routing_tree_to_tables()`* function. A number of routing table minimisation algorithms are also provided to cope with situations where the generated tables do not fit. The *`rig.routing_table.minimise_tables()`* function should prove sufficient for the majority of applications.

Most applications will probably wish to supply the `allocations` optional argument which enables the router to produce `The resource allocated to the resource specified by the ``core_resource` argument (which defaults to *`Cores`*) is assumed to indicate the core number for each vertex.

For example, if a vertex, `v`, is allocated the resources `{Cores: slice(1, 3)}`, if `v` is the sink in a net, that tree will terminate at cores 1 and 2 of the chip `v` is placed on (assuming `core_resource` is *`Cores`*).

Note that if a vertex is allocated an empty set of cores, e.g. `{Cores: slice(0, 0)}`, the tree will terminate at the chip allocated to the vertex but not be routed to any cores.

If the `allocations` argument is ommitted or for any vertices not allocated the `core_resource` resource, the trees produced by this function do not terminate on individual cores but instead terminate on individual chips (with the exception of any constraint-enforced endpoints). The sink vertices are included in the set of children of these *`RoutingTree`* nodes but the route to these children is set to None. It is left up to the application author to decide how to route these vertices in an application-specific post-processing step.

vertices_resources [{vertex: {resource: quantity, ...}, ...}] A dictionary from vertex to the required resources for that vertex. This dictionary must include an entry for every vertex in the application.

Resource requirements are specified by a dictionary *{resource: quantity, ...}* where *resource* is some resource identifier and *quantity* is a non-negative integer representing the quantity of that resource required.

nets [[*`Net`*, ...]] A list (in no particular order) defining the nets connecting vertices.

machine [*`rig.place_and_route.Machine`*] A data structure which defines the resources available in the target SpiNNaker machine.

constraints [[constraint, ...]] A list of constraints on placement, allocation and routing. Available constraints are provided in the *`rig.place_and_route.constraints`* module.

placements [{vertex: (x, y), ...}] A dictionary of the format returned by *`place()`* describing a set of placements of vertices.

> **Warning:** The placement must not have vertices on dead/non-existent chips. failure to comply with this requirement will result in undefined behaviour.

allocations [{vertex: {resource: slice, ...}, ...}] An optional dictionary of the format returned by `allocate()` describing the allocation of resources to vertices. If not supplied, this dictionary defaults to being empty.

core_resource [resource (Default: `Cores`)] **Optional.** Identifier of the resource in *allocations* which indicates the cores to route to when routing to a vertex.

> **Note:** Vertices which do not consume this resource will result in routes which terminate at the chip they're placed on but do not route to any cores.

> **Note:** If no cores are allocated to a vertex, the router will still route the net to the chip where the vertex is placed, but not to any cores.

****kwargs** Additional implementation-specific options.

{*Net*: *RoutingTree*, ...} A dictionary from nets to routing trees which specify an appropriate route through a SpiNNaker machine.

rig.place_and_route.exceptions.InvalidConstraintError If a routing constraint is impossible.

rig.place_and_route.exceptions.MachineHasDisconnectedSubregion If any pair of vertices in a net have no path between them (i.e. the system is impossible to route).

Available algorithms

For more details on the available algorithms, see:

Placement algorithms

`rig.place_and_route.place.sa.`**`place`**(*vertices_resources*, *nets*, *machine*, *constraints*, *effort=1.0*, *random=<module 'random' from '/usr/lib/python2.7/random.pyc'>*, *on_temperature_change=None*, *kernel=<class 'rig.place_and_route.place.sa.python_kernel.PythonKernel'>*, *kernel_kwargs={}*)

A flat Simulated Annealing based placement algorithm.

This placement algorithm uses simulated annealing directly on the supplied problem graph with the objective of reducing wire lengths (and thus, indirectly, the potential for congestion). Though computationally expensive, this placer produces relatively good placement solutions.

The annealing temperature schedule used by this algorithm is taken from "VPR: A New Packing, Placement and Routing Tool for FPGA Research" by Vaughn Betz and Jonathan Rose from the "1997 International Workshop on Field Programmable Logic and Applications".

Two implementations of the algorithm's kernel are available:

- `PythonKernel` A pure Python implementation which is available on all platforms supported by Rig.

- `CKernel` A C implementation which is typically 50-150x faster than the basic Python kernel. Since this implementation requires a C compiler during installation, it is an optional feature of Rig. See the `CKernel's documentation` for details.

The fastest kernel installed is used by default and can be manually chosen using the `kernel` argument.

This algorithm produces INFO level logging information describing the progress made by the algorithm.

Parameters

effort [float] A scaling factor for the number of iterations the algorithm should run for. 1.0 is probably about as low as you'll want to go in practice and runtime increases linearly as you increase this parameter.

random [`random.Random`] A Python random number generator. Defaults to `import random` but can be set to your own instance of `random.Random` to allow you to control the seed and produce deterministic results. For results to be deterministic, vertices_resources must be supplied as an `collections.OrderedDict`.

on_temperature_change [callback_function or None] An (optional) callback function which is called every time the temperature is changed. This callback can be used to provide status updates

The callback function is passed the following arguments:

- `iteration_count`: the number of iterations the placer has attempted (integer)
- `placements`: The current placement solution.
- `cost`: the weighted sum over all nets of bounding-box size. (float)
- `acceptance_rate`: the proportion of iterations which have resulted in an accepted change since the last callback call. (float between 0.0 and 1.0)
- `temperature`: The current annealing temperature. (float)
- `distance_limit`: The maximum distance any swap may be made over. (integer)

If the callback returns False, the anneal is terminated immediately and the current solution is returned.

kernel [`Kernel`] A simulated annealing placement kernel. A sensible default will be chosen based on the available kernels on this machine. The kernel may not be used if the placement problem has a trivial solution.

kernel_kwargs [dict] Optional kernel-specific keyword arguments to pass to the kernel constructor.

`rig.place_and_route.place.rcm.`**`place`**(*vertices_resources*, *nets*, *machine*, *constraints*)

Assigns vertices to chips in Reverse-Cuthill-McKee (RCM) order.

The RCM algorithm (in graph-centric terms) is a simple breadth-first-search-like heuristic which attempts to yield an ordering of vertices which would yield a 1D placement with low network congestion. Placement is performed by sequentially assigning vertices in RCM order to chips, also iterated over in RCM order.

This simple placement scheme is described by Torsten Hoefler and Marc Snir in their paper entitled 'Generic topology mapping strategies for large-scale parallel architectures' published in the Proceedings of the international conference on Supercomputing, 2011.

This is a thin wrapper around the *`sequential`* placement algorithm which uses an RCM ordering for iterating over chips and vertices.

Parameters

breadth_first [bool] Should vertices be placed in breadth first order rather than the iteration order of vertices_resources. True by default.

rig.place_and_route.place.hilbert.**place**(*vertices_resources*, *nets*, *machine*, *constraints*, *breadth_first=True*)

Places vertices in breadth-first order along a hilbert-curve path through the chips in the machine.

This is a thin wrapper around the *sequential* placement algorithm which optionally uses the `breadth_first_vertex_order()` vertex ordering (if the breadth_first argument is True, the default) and `hilbert_chip_order()` for chip ordering.

Parameters

breadth_first [bool] Should vertices be placed in breadth first order rather than the iteration order of vertices_resources. True by default.

rig.place_and_route.place.breadth_first.**place**(*vertices_resources*, *nets*, *machine*, *constraints*, *chip_order=None*)

Places vertices in breadth-first order onto chips in the machine.

This is a thin wrapper around the *sequential* placement algorithm which uses the `breadth_first_vertex_order()` vertex ordering.

Parameters

chip_order [None or iterable] The order in which chips should be tried as a candidate location for a vertex. See the *sequential* placer's argument of the same name.

rig.place_and_route.place.sequential.**place**(*vertices_resources*, *nets*, *machine*, *constraints*, *vertex_order=None*, *chip_order=None*)

Blindly places vertices in sequential order onto chips in the machine.

This algorithm sequentially places vertices onto chips in the order specified (or in an undefined order if not specified). This algorithm is essentially the simplest possible valid placement algorithm and is intended to form the basis of other simple sequential and greedy placers.

The algorithm proceeds by attempting to place each vertex on the a chip. If the vertex fits we move onto the next vertex (but keep filling the same vertex). If the vertex does not fit we move onto the next candidate chip until we find somewhere the vertex fits. The algorithm will raise an `rig.place_and_route.exceptions.InsufficientResourceError` if it has failed to fit a vertex on every chip.

Parameters

vertex_order [None or iterable] The order in which the vertices should be attemted to be placed.

If None (the default), the vertices will be placed in the default iteration order of the `vertices_resources` argument. If an iterable, the iteration sequence should produce each vertex in vertices_resources *exactly once*.

chip_order [None or iterable] The order in which chips should be tried as a candidate location for a vertex.

If None (the default), the chips will be used in the default iteration order of the `machine` object (a raster scan). If an iterable, the iteration sequence should produce (x, y) pairs giving the coordinates of chips to use. All working chip coordinates must be included in the iteration sequence *exactly once*. Additional chip coordinates of non-existant or dead chips are also allowed (and will simply be skipped).

rig.place_and_route.place.rand.**place**(*vertices_resources*, *nets*, *machine*, *constraints*, *random=<module 'random' from '/usr/lib/python2.7/random.pyc'>*)

A random placer.

This algorithm performs uniform-random placement of vertices (completely ignoring connectivty) and thus in the general case is likely to produce very poor quality placements. It exists primarily as a baseline comparison for placement quality and is probably of little value to most users.

Parameters

random [`random.Random`] Defaults to `import random` but can be set to your own instance of `random.Random` to allow you to control the seed and produce deterministic results. For results to be deterministic, vertices_resources must be supplied as an `collections.OrderedDict`.

Allocation algorithms

`rig.place_and_route.allocate.greedy.`**`allocate`**(*vertices_resources*, *nets*, *machine*, *constraints*, *placements*)

Allocate resources to vertices on cores arbitrarily using a simple greedy algorithm.

Routing algorithms

`rig.place_and_route.route.ner.`**`route`**(*vertices_resources*, *nets*, *machine*, *constraints*, *placements*, *allocations={}*, *core_resource=Cores*, *radius=20*)

Routing algorithm based on Neighbour Exploring Routing (NER).

Algorithm refrence: J. Navaridas et al. SpiNNaker: Enhanced multicast routing, Parallel Computing (2014). http://dx.doi.org/10.1016/j.parco.2015.01.002

This algorithm attempts to use NER to generate routing trees for all nets and routes around broken links using A* graph search. If the system is fully connected, this algorithm will always succeed though no consideration of congestion or routing-table usage is attempted.

Parameters

radius [int] Radius of area to search from each node. 20 is arbitrarily selected in the paper and shown to be acceptable in practice. If set to zero, this method is becomes longest dimension first routing.

`constraints`: place and route constraints

Specifications of constraints for placement, allocation and routing.

All constraints defined in this module should be respected by any placement and routing algorithm. Individual algorithms are permitted to define their own implementation-specific constraints seperately.

`class` `rig.place_and_route.constraints.`**`LocationConstraint`**(*vertex*, *location*)

Unconditionally place a vertex on a specific chip.

Attributes

vertex [object] The user-supplied object representing the vertex.

location [(x, y)] The x- and y-coordinates of the chip the vertex must be placed on.

`class` `rig.place_and_route.constraints.`**`SameChipConstraint`**(*vertices*)

Ensure that a group of vertices is always placed on the same chip.

Attributes

vertices [[object, ...]] The list of user-supplied objects representing the vertices to be placed together.

class rig.place_and_route.constraints.**ReserveResourceConstraint**(*resource*, *reservation*, *location=None*)

Reserve a range of a resource on all or a specific chip.

For example, this can be used to reserve areas of SDRAM used by the system software to prevent allocations occurring there.

Note: Reserved ranges must *not* be be partly or fully outside the available resources for a chip nor may they overlap with one another. Violation of these rules will result in undefined behaviour.

Note: placers are obliged by this constraint to subtract the reserved resource from the total available resource but *not* to determine whether the remaining resources include sufficient continuous ranges of resource for their placement. Users should thus be extremely careful reserving resources which are not immediately at the beginning or end of a resource range.

Attributes

resource [object] A resource identifier for the resource being reserved.

reservation [slice] The range over that resource which must not be used.

location [(x, y) or None] The chip to which this reservation applies. If None then the reservation applies globally.

class rig.place_and_route.constraints.**AlignResourceConstraint**(*resource*, *alignment*)

Force alignment of start-indices of resource ranges.

For example, this can be used to ensure assignments into SDRAM are word aligned.

Note: placers are not obliged to be aware of or compensate for wastage of a resource due to this constraint and so may produce impossible placements in the event of large numbers of individual items using a non-aligned width block of resource.

Attributes

resource [object] A resource identifier for the resource to align.

alignment [int] The number of which all assigned start-indices must be a multiple.

class rig.place_and_route.constraints.**RouteEndpointConstraint**(*vertex*, *route*)

Force the endpoint of a path through the network to be a particular route.

This constraint forces routes to/from the constrained vertex to terminate on the route specified in the constraint. For example, this could be used with a vertex representing an external device to force packets sent to the vertex to be absorbed.

Note: This constraint does not check for dead links. This is useful since links attached to external devices will not typically respond to nearest-neighbour PEEK/POKE requests used by the SpiNNaker software to detect link liveness.

Example Usage

If a silicon retina is attached to the north link of chip (1,1) in a 2x2 SpiNNaker machine, the following pair of constraints will ensure traffic destined for the device vertex is routed to the appropriate link:

```
my_device_vertex = ...
constraints = [LocationConstraint(my_device_vertex, (1, 1)),
               RouteEndpointConstraint(my_device_vertex, Routes.north)]
```

Attributes

vertex [object] The user-supplied object representing the vertex.

route [*Routes*] The route to which paths will be directed.

`RoutingTree` data structure

class rig.place_and_route.routing_tree.**RoutingTree**(*chip*, *children=None*)
Explicitly defines a multicast route through a SpiNNaker machine.

Each instance represents a single hop in a route and recursively refers to following steps.

See also:

rig.routing_table.routing_tree_to_tables May be used to convert *RoutingTree* objects into routing tables suitable for loading onto a SpiNNaker machine.

Attributes

chip [(x, y)] The chip the route is currently passing through.

children [list] A `list` of the next steps in the route represented by a (route, object) tuple.

Note: Up until Rig 1.5.1 this structure used `set`s to store children. This was changed to `list`s since sets incur a large memory overhead and in practice the set-like behaviour of the list of children is not useful.

The route must be either *Routes* or *None*. If *Routes* then this indicates the next step in the route uses a particular route.

The object indicates the intended destination of this step in the route. It may be one of:

- *RoutingTree* representing the continuation of the routing tree after following a given link. (Only used if the *Routes* object is a link and not a core).
- A vertex (i.e. some other Python object) when the route terminates at the supplied vertex. Note that the direction may be None and so additional logic may be required to determine what core to target to reach the vertex.

__init__(*chip*, *children=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__iter__()
Iterate over this node and then all its children, recursively and in no specific order. This iterator iterates over the child *objects* (i.e. not the route part of the child tuple).

__repr__() <==> *repr(x)*

traverse()
Traverse the tree yielding the direction taken to a node, the co-ordinates of that node and the directions leading from the Node.

Yields

(direction, (x, y), {:py:class:'~rig.routing_table.Routes', ...}) Direction taken to reach a Node in the tree, the (x, y) co-ordinate of that Node and routes leading to children of the Node.

utils: Utility functions

Utilities functions which assist in the generation of commonly required data structures from the products of placement, allocation and routing.

rig.place_and_route.utils.**build_machine**(*system_info*, *core_resource=Cores*, *sdram_resource=SDRAM*, *sram_resource=SRAM*)

Build a *Machine* object from a *SystemInfo* object.

Note: Links are tested by sending a 'PEEK' command down the link which checks to see if the remote device responds correctly. If the link is dead, no response will be received and the link will be assumed dead. Since peripherals do not generally respond to 'PEEK' commands, working links attached to peripherals will also be marked as dead.

Note: The returned object does not report how much memory is free, nor how many cores are idle but rather the total number of working cores and the size of the heap. See `build_resource_constraints()` for a function which can generate a set of *constraints* which prevent the use of already in-use cores and memory.

Note: This method replaces the deprecated *rig.machine_control.MachineController.get_machine()* method. Its functionality may be recreated using *rig.machine_control.MachineController.get_system_info()* along with this function like so:

```
>> sys_info = mc.get_system_info()
>> machine = build_machine(sys_info)
```

Parameters

system_info [*rig.machine_control.machine_controller.SystemInfo*] The resource availability information for a SpiNNaker machine, typically produced by *rig.machine_control.MachineController.get_system_info()*.

core_resource [resource (default: *rig.place_and_route.Cores*)] The resource type to use to represent the number of working cores on a chip, including the monitor, those already in use and all idle cores.

sdram_resource [resource (default: *rig.place_and_route.SDRAM*)] The resource type to use to represent SDRAM on a chip. This resource will be set to the number of bytes in the largest free block in the SDRAM heap. This gives a conservative estimate of the amount of free SDRAM on the chip which will be an underestimate in the presence of memory fragmentation.

sram_resource [resource (default: *rig.place_and_route.SRAM*)] The resource type to use to represent SRAM (a.k.a. system RAM) on a chip. This resource will be set to the number of bytes in the largest free block in the SRAM heap. This gives a conservative estimate of the amount of free SRAM on the chip which will be an underestimate in the presence of memory fragmentation.

Returns

:py:class:'rig.place_and_route.Machine' A *Machine* object representing the resources available within a SpiNNaker machine in the form used by the place-and-route infrastructure.

rig.place_and_route.utils.**build_core_constraints**(*system_info*, *core_resource=Cores*)

Return a set of place-and-route *ReserveResourceConstraint* which reserve any cores that that are already in use.

The returned list of *ReserveResourceConstraint*s reserves all cores not in an Idle state (i.e. not a monitor and not already running an application).

Note: Historically, every application was required to add a :py:class:~rig.place_and_route.constraints.ReserveResourceConstraint to reserve the monitor processor on each chip. This method improves upon this approach by automatically generating constraints which reserve not just the monitor core but also any other cores which are already in use.

Parameters

system_info [*rig.machine_control.machine_controller.SystemInfo*] The resource availability information for a SpiNNaker machine, typically produced by *rig.machine_control.MachineController.get_system_info()*.

core_resource [resource (Default: *Cores*)] The resource identifier used for cores.

Returns

[:py:class:'rig.place_and_route.constraints.ReserveResourceConstraint', ...] A set of place-and-route constraints which reserves all non-idle cores. The resource type given in the `core_resource` argument will be reserved accordingly.

rig.place_and_route.utils.**build_application_map**(*vertices_applications*, *placements*, *allocations*, *core_resource=Cores*)

Build a mapping from application to a list of cores where the application is used.

This utility function assumes that each vertex is associated with a specific application.

Parameters

vertices_applications [{vertex: application, ...}] Applications are represented by the path of their APLX file.

placements [{vertex: (x, y), ...}]

allocations [{vertex: {resource: slice, ...}, ...}] One of these resources should match the *core_resource* argument.

core_resource [object] The resource identifier which represents cores.

Returns

{application: {(x, y) [set([c, ...]), ...}, ...] For each application, for each used chip a set of core numbers onto which the application should be loaded.

rig.place_and_route.utils.**build_routing_tables**(*routes*, *net_keys*, *omit_default_routes=True*)

DEPRECATED Convert a set of RoutingTrees into a per-chip set of routing tables.

Warning: This method has been deprecated in favour of *rig.routing_table.routing_tree_to_tables()* and *rig.routing_table.minimise()*.

E.g. most applications should use something like:

```
from rig.routing_table import routing_tree_to_tables, minimise
tables = minimise(routing_tree_to_tables(routes, net_keys),
                  target_lengths)
```

Where target_length gives the number of available routing entries on the chips in your SpiNNaker system (see :py:func:~rig.routing_table.utils.build_routing_table_target_lengths)

This command produces routing tables with entries optionally omitted when the route does not change direction (i.e. when default routing can be used).

Warning: A *rig.routing_table.MultisourceRouteError* will be raised if entries with identical keys and masks but with differing routes are generated. This is not a perfect test, entries which would otherwise collide are not spotted.

Warning: The routing trees provided are assumed to be correct and continuous (not missing any hops). If this is not the case, the output is undefined.

Note: If a routing tree has a terminating vertex whose route is set to None, that vertex is ignored.

Parameters

routes [{net: *RoutingTree*, ...}] The complete set of RoutingTrees representing all routes in the system. (Note: this is the same datastructure produced by routers in the *place_and_route* module.)

net_keys [{net: (key, mask), ...}] The key and mask associated with each net.

omit_default_routes [bool] Do not create routing entries for routes which do not change direction (i.e. use default routing).

Returns

{(x, y): [:py:class:'~rig.routing_table.RoutingTableEntry', ...]

Data structures

`Machine` etc.: Machine resource availability P&R data structure

class rig.place_and_route.**Machine**(*width*, *height*, *chip_resources={Cores: 18, SDRAM: 134217728, SRAM: 32768}*, *chip_resource_exceptions={}*, *dead_chips=set([])*, *dead_links=set([])*)

Defines the resources available in a SpiNNaker machine.

This datastructure makes the assumption that in most systems almost everything is uniform and working.

This data-structure intends to be completely transparent. Its contents are described below. A number of utility methods are available but should be considered just that: utilities.

Note: In early versions of Rig this object was called `rig.machine.Machine`.

Attributes

width [int] The width of the system in chips: chips will thus have x-coordinates between 0 and width-1 inclusive.

height [int] The height of the system in chips: chips will thus have y-coordinates between 0 and height-1 inclusive.

chip_resources [{resource_key: requirement, ...}] The resources available on chips (unless otherwise stated in *chip_resource_exceptions). 'resource_key* must be some unique identifying object for a given resource. *requirement* must be a positive numerical value. For example: *{Cores: 17, SDRAM: 128*1024*1024}* would indicate 17 cores and 128 MBytes of SDRAM.

chip_resource_exceptions [{(x,y): resources, ...}] If any chip's resources differ from those specified in *chip_resources*, an entry in this dictionary with the key being the chip's coordinates as a tuple *(x, y)* and *resources* being a dictionary of the same format as *chip_resources*. Note that every exception must specify exactly the same set of keys as *chip_resources*.

dead_chips [set] A set of *(x,y)* tuples enumerating all chips which completely unavailable. Links leaving a dead chip are implicitly marked as dead.

dead_links [set] A set *(x,y,link)* where *x* and *y* are a chip's coordinates and *link* is a value from the Enum *Links*. Note that links have two directions and both should be defined if a link is dead in both directions (the typical case).

__init__(*width*, *height*, *chip_resources={Cores: 18, SDRAM: 134217728, SRAM: 32768}*, *chip_resource_exceptions={}*, *dead_chips=set([])*, *dead_links=set([])*)
Defines the resources available within a SpiNNaker system.

Parameters

width [int]

height [int]

chip_resources [{resource_key: requirement, ...}]

chip_resource_exceptions [{(x,y): resources, ...}]

dead_chips [set([(x,y,p), ...])]

dead_links [set([(x,y,link), ...])]

copy()
Produce a copy of this datastructure.

__eq__(*other*)
Test whether this Machine describes the same machine as another.

__ne__(*other*)
x.__ne__(y) <==> x!=y

issubset(*other*)
Test whether the resources available in this machine description are a (non-strict) subset of those available in another machine.

Note: This test being False does not imply that the this machine is a superset of the other machine; machines may have disjoint resources.

__contains__(*chip_or_link*)
Test if a given chip or link is present and alive.

Parameters

chip_or_link [tuple] If of the form *(x, y, link)*, checks a link. If of the form *(x, y)*, checks a core.

__getitem__(*xy*)
Get the resources available to a given chip.

Raises

IndexError If the given chip is dead or not within the bounds of the system.

__setitem__(*xy*, *resources*)
Specify the resources available to a given chip.

Raises

IndexError If the given chip is dead or not within the bounds of the system.

__iter__()
Iterate over the working chips in the machine.

Generates a series of (x, y) tuples.

__weakref__
list of weak references to the object (if defined)

iter_links()
An iterator over the working links in the machine.

Generates a series of (x, y, link) tuples.

has_wrap_around_links(*minimum_working=0.9*)
Test if a machine has wrap-around connections installed.

Since the Machine object does not explicitly define whether a machine has wrap-around links they must be tested for directly. This test performs a "fuzzy" test on the number of wrap-around links which are working to determine if wrap-around links are really present.

Parameters

minimum_working [0.0 <= float <= 1.0] The minimum proportion of all wrap-around links which must be working for this function to return True.

Returns

bool True if the system has wrap-around links, False if not.

rig.place_and_route.**Cores = Cores**
Resource identifier for (monitor and application) processor cores.

Note that this identifier does not trigger any kind of special-case behaviour in library functions. Users are free to define their own alternatives.

In early versions of Rig this object was called `rig.machine.Cores`.

rig.place_and_route.**SDRAM = SDRAM**
Resource identifier for shared off-die SDRAM (in bytes).

Note that this identifier does not trigger any kind of special-case behaviour in library functions. Users are free to define their own alternatives.

Note: In early versions of Rig this object was called `rig.machine.SDRAM`.

---

rig.place_and_route.**SRAM = SRAM**
Resource identifier for shared on-die SRAM (in bytes).

Note that this identifier does not trigger any kind of special-case behaviour in library functions. Users are free to define their own alternatives.

Note: In early versions of Rig this object was called `rig.machine.SRAM`.

---

rig.links.Links: Chip-to-chip link data structure

class rig.links.**Links**
Enumeration of links from a SpiNNaker chip.

Note that the numbers chosen have two useful properties:

- The integer values assigned are chosen to match the numbers used to identify the links in the low-level software API and hardware registers.
- The links are ordered consecutively in anticlockwise order meaning the opposite link is *(link+3)%6*.

Note: In early versions of Rig this object was called `rig.machine.Links`.

Attributes

east = 0

north_east = 1

north = 2

west = 3

south_west = 4

south = 5

rig.netlist.Net: Net data structure

class rig.netlist.**Net**(*source*, *sinks*, *weight=1.0*)
A net represents connectivity from one vertex to many vertices.

Attributes

source [vertex] The vertex which is the source of the net.

weight [float or int] The "strength" of the net, in application specific units.

sinks [list] A list of vertices that the net connects to.

__init__(*source*, *sinks*, *weight=1.0*)
Create a new Net.

Parameters

source [vertex]

sinks [list or vertex] If a list of vertices is provided then the list is copied, whereas if a single vertex is provided then this used to create the list of sinks.

weight [float or int]

__contains__(*vertex*)
Test if a supplied vertex is a source or sink of this net.

__iter__()
Iterate over all vertices in the net, starting with the source.

__weakref__
list of weak references to the object (if defined)

2.2.2 `rig.routing_table`: Multicast routing table datastructures and tools

This module contains data structures and algorithms for representing and manipulating multicast routing tables for SpiNNaker.

Quick-start Examples

The following examples give quick examples of Rig's routing table data structures and table minimisation tools.

Using the place-and-route wrapper

If you're using the *place_and_route_wrapper()* wrapper function to perform place-and-route for your application, routing table minimisation is performed automatically when required. No changes are required to your application!

Manually defining and minimising individual routing tables

The brief example below illustrates how a single routing table might be defined and minimised.

```
>>> # Define a (trivially minimised) example routing table
>>> from rig.routing_table import Routes, RoutingTableEntry
>>> original = [
...     RoutingTableEntry({Routes.north}, 0x00000000, 0xFFFFFFFF),
...     RoutingTableEntry({Routes.north}, 0x00000001, 0xFFFFFFFF),
...     RoutingTableEntry({Routes.north}, 0x00000002, 0xFFFFFFFF),
...     RoutingTableEntry({Routes.north}, 0x00000003, 0xFFFFFFFF),
... ]

>>> # Minimise the routing table using a sensible selection of algorithms
>>> from rig.routing_table import minimise_table
>>> minimised = minimise_table(original, target_length=None)
>>> assert minimised == [
...     RoutingTableEntry({Routes.north}, 0x00000000, 0xFFFFFFFC),
... ]
```

Generating and loading routing tables from automatic place-and-route tools

The outline below shows how routing tables might be generated from the results of Rig's *place and route* tools, minimised and then loaded onto a SpiNNaker machine.

```
# Interrogate the SpiNNaker machine to determine what resources are
# available (including the number of multicast routing table entries on
# each chip).
from rig.machine_control import MachineController
machine_controller = MachineController("hostname")
system_info = machine_controller.get_system_info()

# Place-and-route your application as normal and select suitable
# routing keys for each net.
from rig.place_and_route import route
routes = route(...)
net_keys = {Net: (key, mask), ...}

# Produce routing tables from the generated routes
from rig.routing_table import routing_tree_to_tables
routing_tables = routing_tree_to_tables(routes, net_keys)

# Minimise the routes (if required), trying a sensible selection of table
# minimisation algorithms.
from rig.routing_table import (
    build_routing_table_target_lengths,
    minimise_tables)
target_lengths = build_routing_table_target_lengths(system_info)
routing_tables = minimise_tables(routing_tables, target_lengths)

# Load the minimised routing tables onto SpiNNaker
machine_controller.load_routing_tables(routing_tables)
```

`RoutingTableEntry` and `Routes`: Routing table data structures

Routing tables in Rig are conventionally represented as a list of *`RoutingTableEntry`* objects in the order they would appear in a SpiNNaker router. Empty/unused routing table entries are not usually included in these representations.

class rig.routing_table.**RoutingTableEntry**

Named tuple representing a single routing entry in a SpiNNaker routing table.

Parameters

route [{*`Routes`*, ...}] The set of destinations a packet should be routed to where each element in the set is a value from the enumeration *`Routes`*.

key [int] 32-bit unsigned integer routing key to match after applying the mask.

mask [int] 32-bit unsigned integer mask to apply to keys of packets arriving at the router.

sources [{*`Routes`*, ...}] Links on which a packet may enter the router before taking this route. If the source directions are unknown `{None}` should be used (the default).

class rig.routing_table.**Routes**

Enumeration of routes which a SpiNNaker packet can take after arriving at a router.

Note that the integer values assigned are chosen to match the numbers used to identify routes in the low-level software API and hardware registers.

Note that you can directly cast from a *`rig.links.Links`* to a Routes value.

Attributes

east = 0

north_east = 1

north = 2

west = 3

south_west = 4

south = 5

core_monitor = 6

core_1 = 7

core_2 = 8

core_3 = 9

core_4 = 10

core_5 = 11

core_6 = 12

core_7 = 13

core_8 = 14

core_9 = 15

core_10 = 16

core_11 = 17

core_12 = 18

core_13 = 19

core_14 = 20

core_15 = 21

core_16 = 22

core_17 = 23

Routing table construction utility

The *routing_tree_to_tables()* function is provided which constructs routing tables of the form described above from *RoutingTree* objects produced by an automatic routing algorithm.

rig.routing_table.**routing_tree_to_tables**(*routes*, *net_keys*)
Convert a set of *RoutingTree* s into a per-chip set of routing tables.

> **Warning:** A *rig.routing_table.MultisourceRouteError* will be raised if entries with identical keys and masks but with differing routes are generated. This is not a perfect test, entries which would otherwise collide are not spotted.

> **Warning:** The routing trees provided are assumed to be correct and continuous (not missing any hops). If this is not the case, the output is undefined.

Note: If a routing tree has a terminating vertex whose route is set to None, that vertex is ignored.

Parameters

routes [{net: *RoutingTree*, ...}] The complete set of RoutingTrees representing all routes in the system. (Note: this is the same data structure produced by routers in the *place_and_route* module.)

net_keys [{net: (key, mask), ...}] The key and mask associated with each net.

Returns

{(x, y): [:py:class:'~rig.routing_table.RoutingTableEntry', ...]

exception rig.routing_table.**MultisourceRouteError**(*key*, *mask*, *coordinate*)

Indicates that two nets with the same key and mask would cause packets to become duplicated.

Routing table minimisation algorithms

SpiNNaker's multicast routing tables are a finite resource containing a maximum of 1024 entries. Certain applications may find that they exhaust this limited resource and may wish to attempt to shrink their routing tables by making better use of the SpiNNaker router's capabilities. For example, if a packet's key does not match any routing entries it will be "default routed" in the direction in which it was already travelling and thus no routing table entry is required. Additionally, by more fully exploiting the behaviour of the Ternary Content Addressable Memory (TCAM) used in SpiNNaker's multicast router it is often possible to compress (or minimise) a given routing table into a more compact, yet logically equivalent, form.

This module includes algorithms for minimising routing tables for use by SpiNNaker application developers.

Common-case wrappers

For most users, the following functions can be used to minimise routing tables used by their application. Both accept a target number of routing entries and will attempt to apply routing table minimisation algorithms from this module until the supplied tables fit.

rig.routing_table.**minimise_tables**(*routing_tables*, *target_lengths*, *methods=(<function minimise>*, *<function minimise>)*)

Utility function which attempts to minimises routing tables for multiple chips.

For each routing table supplied, this function will attempt to use the minimisation algorithms given (or some sensible default algorithms), trying each sequentially until a target number of routing entries has been reached.

Parameters

routing_tables [{(x, y): [*RoutingTableEntry*, ...], ...}] Dictionary mapping chip co-ordinates to the routing tables associated with that chip. NOTE: This is the data structure as returned by *routing_tree_to_tables()*.

target_lengths [int or {(x, y): int or None, ...} or None] Maximum length of routing tables. If an integer this is assumed to be the maximum length for any table; if a dictionary then it is assumed to be a mapping from co-ordinate to maximum length (or None); if None then tables will be minimised as far as possible.

methods : Each method is tried in the order presented and the first to meet the required target length for a given chip is used. Consequently less computationally costly algorithms

should be nearer the start of the list. The defaults will try to remove default routes (*rig.routing_table.remove_default_routes.minimise()*) and then fall back on the ordered covering algorithm (*rig.routing_table.ordered_covering.minimise()*).

Returns

{(x, y): [:py:class:'~rig.routing_table.RoutingTableEntry', ...], ...} Minimised routing tables, guaranteed to be at least as small as the table sizes specified by *target_lengths*.

Raises

MinimisationFailedError If no method can sufficiently minimise a table.

rig.routing_table.**minimise_table**(*table*, *target_length*, *methods=(<function minimise>, <function minimise>)*)

Apply different minimisation algorithms to minimise a single routing table.

Parameters

table [[*RoutingTableEntry*, ...]] Routing table to minimise. NOTE: This is the data structure as returned by *routing_tree_to_tables()*.

target_length [int or None] Maximum length of the routing table. If None then all methods will be tried and the smallest achieved table will be returned.

methods : Each method is tried in the order presented and the first to meet the required target length for a given chip is used. Consequently less computationally costly algorithms should be nearer the start of the list. The defaults will try to remove default routes (:py:meth:rig.routing_table.remove_default_routes.minimise) and then fall back on the ordered covering algorithm (:py:meth:rig.routing_table.ordered_covering.minimise).

Returns

[:py:class:'~rig.routing_table.RoutingTableEntry', ...] Minimised routing table, guaranteed to be at least as small as *target_length*, or as small as possible if *target_length* is None.

Raises

MinimisationFailedError If no method can sufficiently minimise the table.

Available algorithms

The following minimisation algorithms are currently available:

Remove Default Routes

rig.routing_table.remove_default_routes.**minimise**(*table*, *target_length*, *check_for_aliases=True*)

Remove from the routing table any entries which could be replaced by default routing.

Parameters

routing_table [[*RoutingTableEntry*, ...]] Routing table from which to remove entries which could be handled by default routing.

target_length [int or None] Target length of the routing table.

check_for_aliases [bool] If True (the default), default-route candidates are checked for aliased entries before suggesting a route may be default routed. This check is required to ensure

correctness in the general case but has a runtime complexity of O(N^2) in the worst case for N-entry tables.

If False, the alias-check is skipped resulting in O(N) runtime. This option should only be used if the supplied table is guaranteed not to contain any aliased entries.

Returns

[:py:class:'~rig.routing_table.RoutingTableEntry', ...] Reduced routing table entries.

Raises

MinimisationFailedError If the smallest table that can be produced is larger than *target_length*.

Ordered Covering

An novel algorithm for the minimisation of SpiNNaker's multicast routing tables devised by Andrew Mundy.

Background

SpiNNaker routing tables consist of entries made up of a 32-bit key, a 32-bit mask and a 24-bit route value. The key and mask of every entry act as a sieve for the keys found on incoming multicast packets. Each bit of the key-mask pair can be considered as matching 0, 1 or 2 values in the same bit of a multicast packet key:

Key	Mask	Matches Key Values	Written
0	0	0 or 1	X
0	1	0	0
1	1	1	1
1	0	Nothing	!

If a packet matches the key-mask of an entry then the packet is transmitted to the cores and links indicated by the route field.

For example, if the table were:

Key	Mask	Route
0000	1111	North, North East
0111	0111	South

Which, from now on, will be written as:

```
0000 -> N NE
X111 -> S
```

Then any packets with the key 0000 would be sent out of the north and north-east links. Any packets with the keys 0111 or 1111 would be sent out of the south link only.

Entries in table are ordered, with entries at the top of the table having higher priority than those lower down the table. Only the highest priority entry which matches a packet is used. If, for example, the table were:

```
0000 -> N NE
1111 -> 1 2
X111 -> S
```

Then packets with the keys `0000` and `0111` would be treated as before. However, packets with the key `1111` would be sent to cores 1 and 2 as only the higher priority entry has effect.

Merging routing table entries

Routing tables can be minimised by merging together entries with equivalent routes. This is done by creating a new key-mask pair with an `X` wherever the key-mask pairs of any of the original entries differed.

For example, merging of the entries:

```
0000 -> N
0001 -> N
```

Would lead to the new entry:

000X -> N

Which would match any of the keys matched by the original entries but no more. In contrast the merge of `0001` and `0010` would generate the new entry `00XX` which would match keys matched by either of the original entries but also `0000` and `0011`.

Clearly, if we are to attempt to minimise tables such as:

```
0001 -> N
0010 -> N
0000 -> S, SE
0011 -> SE
```

We need a set of rules for:

1. Where merged entries are to be inserted into the table
2. Which merges are allowed

"Ordered Covering"

The algorithm implemented here, "Ordered Covering", provides the following rule:

- The only merges allowed are those which:
    1. would not cause one of the entries in the merge to be "hidden" below an entry of lesser generality than the merged entry but which matched any of the same keys. For example, merging `0010` and `0001` would not be allowed if the new entry would be placed below the existing entry `000X` as this would "hide" `0001`.
    2. would not cause an entry "contained" within an entry of higher generality to be hidden by the insertion of a new entry. For example, if the entry `XXXX` had been formed by merging the entries `0011` and `1100` then merging of the entries `1101` and `1110` would not be allowed as it would cause the entry `11XX` to be inserted above `XXXX` in the table and would hide `1100`.

Following these rules ensures that the minimised table will be functionally equivalent to the original table provided that the original table was invariant under reordering OR was provided in increasing order of generality.

As a heuristic:

- Routing tables are to be kept sorted in increasing order of "generality", that is the number of `X``s in the entry. An entry with the key-mask pair ``00XX` must be placed below any entries with fewer `X``s in their key-mask pairs (e.g., below ``0000` and `000X`).

1. New entries must also be inserted below any entries of the same generality. If `XX00` were already present in the table the new entry `0XX0` must be inserted below it.

rig.routing_table.ordered_covering.**minimise**(*routing_table*, *target_length*)

Reduce the size of a routing table by merging together entries where possible and by removing any remaining default routes.

Warning: The input routing table *must* also include entries which could be removed and replaced by default routing.

Warning: It is assumed that the input routing table is not in any particular order and may be reordered into ascending order of generality (number of don't cares/Xs in the key-mask) without affecting routing correctness. It is also assumed that if this table is unordered it is at least orthogonal (i.e., there are no two entries which would match the same key) and reorderable.

Note: If *all* the keys in the table are derived from a single instance of *BitField* then the table is guaranteed to be orthogonal and reorderable.

Note: Use *expand_entries()* to generate an orthogonal table and receive warnings if the input table is not orthogonal.

Parameters

routing_table [[*RoutingTableEntry*, ...]] Routing entries to be merged.

target_length [int or None] Target length of the routing table; the minimisation procedure will halt once either this target is reached or no further minimisation is possible. If None then the table will be made as small as possible.

Returns

[:py:class:'~rig.routing_table.RoutingTableEntry', ...] Reduced routing table entries.

Raises

MinimisationFailedError If the smallest table that can be produced is larger than *target_length*.

rig.routing_table.ordered_covering.**ordered_covering**(*routing_table*, *target_length*, *aliases={}*, *no_raise=False*)

Reduce the size of a routing table by merging together entries where possible.

Warning: The input routing table *must* also include entries which could be removed and replaced by default routing.

Warning: It is assumed that the input routing table is not in any particular order and may be reordered into ascending order of generality (number of don't cares/Xs in the key-mask) without affecting routing

correctness. It is also assumed that if this table is unordered it is at least orthogonal (i.e., there are no two entries which would match the same key) and reorderable.

Note: If *all* the keys in the table are derived from a single instance of *BitField* then the table is guaranteed to be orthogonal and reorderable.

Note: Use *expand_entries()* to generate an orthogonal table and receive warnings if the input table is not orthogonal.

Parameters

routing_table [[*RoutingTableEntry*, ...]] Routing entries to be merged.

target_length [int or None] Target length of the routing table; the minimisation procedure will halt once either this target is reached or no further minimisation is possible. If None then the table will be made as small as possible.

Returns

[:py:class:'~rig.routing_table.RoutingTableEntry', ...] Reduced routing table entries.

{(key, mask): {(key, mask), ...}, ...} A new aliases dictionary.

Other Parameters

aliases [{(key, mask): {(key, mask), ...}, ...}] Dictionary of which keys and masks in the routing table are combinations of other (now removed) keys and masks; this allows us to consider only the keys and masks the user actually cares about when determining if inserting a new entry will break the correctness of the table. This should be supplied when using this method to update an already minimised table.

no_raise [bool] If False (the default) then an error will be raised if the table cannot be minimised to be smaller than *target_length* and *target_length* is not None. If True then a table will be returned regardless of the size of the final table.

Raises

MinimisationFailedError If the smallest table that can be produced is larger than *target_length* and *no_raise* is False.

`minimise()` prototype

Routing table minimisation functions are always named `minimise()` and are contained within a Python module named after the algorithm. These *minimise()* functions have the signature defined below.

`rig.routing_table.`**`minimise`**(*routing_table*, *target_length=1024*)

Reduce the size of a routing table by merging together entries where possible.

Warning: The input routing table *must* also include entries which could be removed and replaced by default routing.

Warning: It is assumed that the input routing table is not in any particular order and may be reordered into ascending order of generality (number of don't cares/Xs in the key-mask) without affecting routing correctness. It is also assumed that if this table is unordered it is at least orthogonal (i.e., there are no two entries which would match the same key) and reorderable.

Note: If *all* the keys in the table are derived from a single instance of `BitField` then the table is guaranteed to be orthogonal and reorderable.

Note: Use `expand_entries()` to generate an orthogonal table and receive warnings if the input table is not orthogonal.

routing_table [[`RoutingTableEntry`, ...]] Routing entries to be merged.

target_length [int or None] If an int, this is the target length of the routing table; the minimisation procedure may halt once either this target is reached or no further minimisation is possible. If the target could not be reached a `MinimisationFailedError` will be raised.

If None then the table will be made as small as possible and is guaranteed to return a result.

MinimisationFailedError If the smallest table that can be produced is larger than `target_length` and `target_length` is not None.

[`RoutingTableEntry`, ...] Reduced routing table entries. The returned routing table is guaranteed to route all entries matched by the input table in the same way. Note that the minimised table may also match keys *not* previously matched by the input routing table.

exception rig.routing_table.**MinimisationFailedError**(*target_length*, *final_length=None*, *chip=None*)

Raised when a routing table could not be minimised to reach a specified target.

Attributes

target_length [int] The target number of routing entries.

final_length [int] The number of routing entries reached when the algorithm completed. (final_length > target_length)

chip [(x, y) or None] The coordinates of the chip on which routing table minimisation first failed. Only set when minimisation is performed across many chips simultaneously.

Routing Table Manipulation Tools

The following functions may be useful when comparing routing tables, for example if testing or evaluating minimisation algorithms.

rig.routing_table.**table_is_subset_of**(*entries_a*, *entries_b*)

Check that every key matched by every entry in one table results in the same route when checked against the other table.

For example, the table:

```
>>> from rig.routing_table import Routes
>>> table = [
...     RoutingTableEntry({Routes.north, Routes.north_east}, 0x0, 0xf),
...     RoutingTableEntry({Routes.east}, 0x1, 0xf),
...     RoutingTableEntry({Routes.south_west}, 0x5, 0xf),
...     RoutingTableEntry({Routes.north, Routes.north_east}, 0x8, 0xf),
...     RoutingTableEntry({Routes.east}, 0x9, 0xf),
...     RoutingTableEntry({Routes.south_west}, 0xe, 0xf),
...     RoutingTableEntry({Routes.north, Routes.north_east}, 0xc, 0xf),
...     RoutingTableEntry({Routes.south, Routes.south_west}, 0x0, 0xb),
... ]
```

is a functional subset of a minimised version of itself:

```
>>> from rig.routing_table.ordered_covering import minimise
>>> other_table = minimise(table, target_length=None)
>>> other_table == table
False
>>> table_is_subset_of(table, other_table)
True
```

But not vice-versa:

```
>>> table_is_subset_of(other_table, table)
False
```

Default routes are taken into account, such that the table:

```
>>> table = [
...     RoutingTableEntry({Routes.north}, 0x0, 0xf, {Routes.south}),
... ]
```

is a subset of the empty table:

```
>>> table_is_subset_of(table, list())
True
```

Parameters

entries_a [[*RoutingTableEntry*, ...]]

entries_b [[*RoutingTableEntry*, ...]] Ordered of lists of routing table entries to compare.

Returns

bool True if every key matched in *entries_a* would result in an equivalent route for the packet when matched in *entries_b*.

rig.routing_table.**expand_entries**(*entries*, *ignore_xs=None*)

Turn all Xs which are not ignored in all entries into `0` s and `1` s.

For example:

```
>>> from rig.routing_table import RoutingTableEntry
>>> entries = [
...     RoutingTableEntry(set(), 0b0100, 0xfffffff0 | 0b1100),  # 01XX
...     RoutingTableEntry(set(), 0b0010, 0xfffffff0 | 0b0010),  # XX1X
... ]
```

(continues on next page)

(continued from previous page)

```
>>> list(expand_entries(entries)) == [
...     RoutingTableEntry(set(), 0b0100, 0xffffffff | 0b1110),  # 010X
...     RoutingTableEntry(set(), 0b0110, 0xffffffff | 0b1110),  # 011X
...     RoutingTableEntry(set(), 0b0010, 0xffffffff | 0b1110),  # 001X
...     RoutingTableEntry(set(), 0b1010, 0xffffffff | 0b1110),  # 101X
...     RoutingTableEntry(set(), 0b1110, 0xffffffff | 0b1110),  # 111X
... ]
True
```

Note that the `X` in the LSB was retained because it is common to all entries.

Any duplicated entries will be removed (in this case the first and second entries will both match `0000`, so when the second entry is expanded only one entry is retained):

```
>>> from rig.routing_table import Routes
>>> entries = [
...     RoutingTableEntry({Routes.north}, 0b0000, 0b1111),  # 0000 -> N
...     RoutingTableEntry({Routes.south}, 0b0000, 0b1011),  # 0X00 -> S
... ]
>>> list(expand_entries(entries)) == [
...     RoutingTableEntry({Routes.north}, 0b0000, 0b1111),  # 0000 -> N
...     RoutingTableEntry({Routes.south}, 0b0100, 0b1111),  # 0100 -> S
... ]
True
```

Warning: It is assumed that the input routing table is orthogonal (i.e., there are no two entries which would match the same key). If this is not the case, any entries which are covered (i.e. unreachable) in the input table will be omitted and a warning produced. As a result, all output routing tables are guaranteed to be orthogonal.

Parameters

entries [[*RoutingTableEntry*...] or similar] The entries to expand.

Yields

:py:class:‘~rig.routing_table.RoutingTableEntry‘ Routing table entries which represent the original entries but with all Xs not masked off by *ignore_xs* replaced with 1s and 0s.

Other Parameters

ignore_xs [int] Mask of bits in which Xs should not be expanded. If None (the default) then Xs which are common to all entries will not be expanded.

`rig.routing_table.`**`intersect`**(*key_a*, *mask_a*, *key_b*, *mask_b*)

Return if key-mask pairs intersect (i.e., would both match some of the same keys).

For example, the key-mask pairs `00XX` and `001X` both match the keys `0010` and `0011` (i.e., they do intersect):

```
>>> intersect(0b0000, 0b1100, 0b0010, 0b1110)
True
```

But the key-mask pairs `00XX` and `11XX` do not match any of the same keys (i.e., they do not intersect):

```
>>> intersect(0b0000, 0b1100, 0b1100, 0b1100)
False
```

Parameters

key_a [int]

mask_a [int] The first key-mask pair

key_b [int]

mask_b [int] The second key-mask pair

Returns

bool True if the two key-mask pairs intersect otherwise False.

Utility Functions

rig.routing_table.**build_routing_table_target_lengths**(*system_info*)
Build a dictionary of target routing table lengths from a *SystemInfo* object.

Useful in conjunction with *minimise_tables()*.

Returns

{(x, y): num, ...} A dictionary giving the number of free routing table entries on each chip on a SpiNNaker system.

Note: The actual number of entries reported is the size of the largest contiguous free block of routing entries in the routing table.

2.2.3 rig.geometry: Machine geometry utility functions

General-purpose SpiNNaker-related geometry functions.

rig.geometry.**to_xyz**(*xy*)
Convert a two-tuple (x, y) coordinate into an (x, y, 0) coordinate.

rig.geometry.**minimise_xyz**(*xyz*)
Minimise an (x, y, z) coordinate.

rig.geometry.**shortest_mesh_path_length**(*source*, *destination*)
Get the length of a shortest path from source to destination without using wrap-around links.

Parameters

source [(x, y, z)]

destination [(x, y, z)]

Returns

int

rig.geometry.**shortest_mesh_path**(*source*, *destination*)
Calculate the shortest vector from source to destination without using wrap-around links.

Parameters

source [(x, y, z)]

destination [(x, y, z)]

Returns

(x, y, z)

rig.geometry.**shortest_torus_path_length**(*source*, *destination*, *width*, *height*)
Get the length of a shortest path from source to destination using wrap-around links.

See http://jhnet.co.uk/articles/torus_paths for an explanation of how this method works.

Parameters

source [(x, y, z)]

destination [(x, y, z)]

width [int]

height [int]

Returns

int

rig.geometry.**shortest_torus_path**(*source*, *destination*, *width*, *height*)
Calculate the shortest vector from source to destination using wrap-around links.

See http://jhnet.co.uk/articles/torus_paths for an explanation of how this method works.

Note that when multiple shortest paths exist, one will be chosen at random with uniform probability.

Parameters

source [(x, y, z)]

destination [(x, y, z)]

width [int]

height [int]

Returns

(x, y, z)

rig.geometry.**concentric_hexagons**(*radius*, *start=(0, 0)*)
A generator which produces coordinates of concentric rings of hexagons.

Parameters

radius [int] Number of layers to produce (0 is just one hexagon)

start [(x, y)] The coordinate of the central hexagon.

rig.geometry.**standard_system_dimensions**(*num_boards*)
Calculate the standard network dimensions (in chips) for a full torus system with the specified number of SpiNN-5 boards.

Returns

(w, h) Width and height of the network in chips.

Standard SpiNNaker systems are constructed as squarely as possible given the number of boards available. When a square system cannot be made, the function prefers wider systems over taller systems.

Raises

ValueError If the number of boards is not a multiple of three.

rig.geometry.**spinn5_eth_coords**(*width*, *height*, *root_x=0*, *root_y=0*)
Generate a list of board coordinates with Ethernet connectivity in a SpiNNaker machine.

Specifically, generates the coordinates for the Ethernet connected chips of SpiNN-5 boards arranged in a standard torus topology.

Warning: In general, applications should use *rig.machine_control.MachineController.get_system_info* and *ethernet_connected_chips()* to gather the coordinates of Ethernet connected chips which are actually functioning. For example:

```
>> from rig.machine_control import MachineController
>> mc = MachineController("my-machine")
>> si = mc.get_system_info()
>> print(list(si.ethernet_connected_chips()))
[((0, 0), "1.2.3.4"), ((4, 8), "1.2.3.5"), ((8, 4), "1.2.3.6")]
```

Parameters

width, height [int] Width and height of the system in chips.

root_x, root_y [int] The coordinates of the root chip (i.e. the chip used to boot the machine), e.g. from *rig.machine_control.MachineController.root_chip*.

rig.geometry.**spinn5_local_eth_coord**(*x*, *y*, *w*, *h*, *root_x=0*, *root_y=0*)
Get the coordinates of a chip's local ethernet connected chip.

Returns the coordinates of the ethernet connected chip on the same board as the supplied chip.

Note: This function assumes the system is constructed from SpiNN-5 boards

Warning: In general, applications should interrogate the machine to determine which Ethernet connected chip is considered 'local' to a particular SpiNNaker chip, e.g. using *rig.machine_control.MachineController.get_system_info*:

```
>> from rig.machine_control import MachineController
>> mc = MachineController("my-machine")
>> si = mc.get_system_info()
>> print(si[(3, 2)].local_ethernet_chip)
(0, 0)
```

spinn5_local_eth_coord() will always produce the coordinates of the Ethernet-connected SpiNNaker chip on the same SpiNN-5 board as the supplied chip. In future versions of the low-level system software, some other method of choosing local Ethernet connected chips may be used.

Parameters

x, y [int] Chip whose coordinates are of interest.

w, h [int] Width and height of the system in chips.

root_x, root_y [int] The coordinates of the root chip (i.e. the chip used to boot the machine), e.g. from *rig.machine_control.MachineController.root_chip*.

rig.geometry.**SPINN5_ETH_OFFSET = array([[[0, 0], [-1, 0], [-2, 0], [-3, 0], [-4, 0], [-1,**
SpiNN-5 ethernet connected chip lookup.

Used by *spinn5_local_eth_coord()*. Given an x and y chip position modulo 12, return the offset of the board's bottom-left chip from the chip's position.

Note: the order of indexes: `SPINN5_ETH_OFFSET[y][x]`!

rig.geometry.**spinn5_chip_coord**(*x*, *y*, *root_x=0*, *root_y=0*)
Get the coordinates of a chip on its board.

Given the coordinates of a chip in a multi-board system, calculates the coordinates of the chip within its board.

Note: This function assumes the system is constructed from SpiNN-5 boards

---

Parameters

x, y [int] The coordinates of the chip of interest

root_x, root_y [int] The coordinates of the root chip (i.e. the chip used to boot the machine), e.g. from *rig.machine_control.MachineController.root_chip*.

rig.geometry.**spinn5_fpga_link**(*x*, *y*, *link*, *root_x=0*, *root_y=0*)
Get the identity of the FPGA link which corresponds with the supplied link.

Note: This function assumes the system is constructed from SpiNN-5 boards whose FPGAs are loaded with the SpI/O 'spinnaker_fpgas' image.

Parameters

x, y [int] The chip whose link is of interest.

link [Link] The link of interest.

root_x, root_y [int] The coordinates of the root chip (i.e. the chip used to boot the machine), e.g. from *rig.machine_control.MachineController.root_chip*.

Returns

(fpga_num, link_num) or None If not None, the link supplied passes through an FPGA link. The returned tuple indicates the FPGA responsible for the sending-side of the link.

fpga_num is the number (0, 1 or 2) of the FPGA responsible for the link.

link_num indicates which of the sixteen SpiNNaker links (0 to 15) into an FPGA is being used. Links 0-7 are typically handled by S-ATA link 0 and 8-15 are handled by S-ATA link 1.

Returns None if the supplied link does not pass through an FPGA.

rig.geometry.**SPINN5_FPGA_LINKS = {(0, 0, <Links.west: 3>): (1, 1), (0, 0, <Links.south_we**
FPGA link IDs for each link leaving a SpiNN-5 board.

Format:

```
{(x, y, link): (fpga_num, link_num), ...}
```

Used by *spinn5_fpga_link()*.

2.3 Execution control and machine management

2.3.1 `rig.machine_control`: Machine Control APIs

Rig provides various high-level APIs for communicating with and controlling SpiNNaker machines. New users are encouraged to start by working through the *introductory tutorials*:

`MachineController`: SpiNNaker Control API

class rig.machine_control.**MachineController**(*initial_host*, *scp_port=17893*, *boot_port=54321*, *n_tries=5*, *timeout=0.5*, *structs=None*, *initial_context={'app_id': 66}*)

A high-level interface for controlling a SpiNNaker system.

This class is essentially a wrapper around key functions provided by the SCP protocol which aims to straightforwardly handle many of the difficult details and corner cases to ensure easy, efficient and reliable communication with and control of a SpiNNaker machine. A *tutorial* is available for new users.

Key features at a glance:

- Machine booting
- Probing for available resources
- (Efficient & reliable) loading of applications
- Application monitoring and control
- Allocation and loading of routing tables
- Allocation and loading of memory
- An optional file-like interface to memory blocks
- Setting up IPTags
- Easy-to-use blocking API

Coming soon:

- (Additional) 'advanced' non-blocking, parallel I/O interface
- (Automagically) handling multiple connections simultaneously

This class does *not* provide any methods for sending and receiving arbitrary SDP packets to and from applications. For this you should use `sockets` and the *`rig.machine_control.packets`* library (for which a *tutorial* is also available).

This class features a context system which allows commonly required arguments to be specified for a whole block of code using a 'with' statement, for example:

```
cm = MachineController("spinnaker")

# Commands should refer to chip (2, 3)
with cm(x=2, y=3):
    three_kb_of_joy = cm.sdram_alloc(3*1024)
    cm.write(three_kb_of_joy, b"joy" * 1024)
    core_one_status = cm.get_processor_status(1)
```

__init__(*initial_host*, *scp_port=17893*, *boot_port=54321*, *n_tries=5*, *timeout=0.5*, *structs=None*, *initial_context={'app_id': 66}*)
Create a new controller for a SpiNNaker machine.

Parameters

initial_host [string] Hostname or IP address of the SpiNNaker chip to connect to. If the board has not yet been booted, this will be used to boot the machine.

scp_port [int] Port number for SCP connections.

boot_port [int] Port number for booting the board.

n_tries [int] Number of SDP packet retransmission attempts.

timeout [float] Timeout in seconds before an SCP response is assumed lost and the request is retransmitted.

structs [dict or None] A dictionary of struct data defining the memory locations of important values in SARK as produced by *rig.machine_control.struct_file.read_struct_file*. If None, the default struct file will be used.

initial_context [*{argument: value}*] Default argument values to pass to methods in this class. By default this just specifies a default App-ID.

__call__(***context_args*)
For use with *with*: set default argument values.

E.g:

```
with controller(x=3, y=4):
    # All commands in this block now communicate with chip (3, 4)
```

scp_data_length
The maximum SCP data field length supported by the machine (bytes).

scp_window_size
The maximum number of packets that can be sent to a SpiNNaker board without receiving any acknowledgement packets.

root_chip
The coordinates (x, y) of the chip used to boot the machine.

send_scp(**args*, *y=Required*, *x=Required*, *p=Required*, ***kwargs*)
Transmit an SCP Packet and return the response.

This function is a thin wrapper around *rig.machine_control.scp_connection.SCPConnection.send_scp()*.

This function will attempt to use the SCP connection nearest the destination of the SCP command if multiple connections have been discovered using *discover_connections()*.

Parameters

x [int]

y [int]

p [int]

***args**

****kwargs**

boot(*width=None*, *height=None*, *only_if_needed=True*, *check_booted=True*, ***boot_kwargs*)
Boot a SpiNNaker machine.

The system will be booted from the Ethernet connected chip whose hostname was given as the argument to the MachineController. With the default arguments this method will only boot systems which have not already been booted and will wait until machine is completely booted (and raise a *SpiNNakerBootError* on failure).

This method uses *rig.machine_control.boot.boot()* to send boot commands to the machine and update the struct files contained within this object according to those used during boot.

Warning: Booting the system over the open internet is likely to fail due to the port number being blocked by most ISPs and UDP not being reliable. A proxy such as spinnaker_proxy may be useful in this situation.

Parameters

width, height [*Deprecated*] **Deprecated.** In older versions of SC&MP, it was necessary to indicate the size of the machine being booted. These parameters are now ignored and setting them will produce a deprecation warning.

scamp_binary [filename or None] Filename of the binary to boot the machine with or None to use the SC&MP binary bundled with Rig.

sark_struct [filename or None] The 'sark.struct' file which defines the datastructures or None to use the one bundled with Rig.

boot_delay [float] Number of seconds to pause between sending boot data packets.

post_boot_delay [float] Number of seconds to wait after sending last piece of boot data to give SC&MP time to re-initialise the Ethernet interface.

only_if_needed [bool] If `only_if_needed` is True (the default), this method checks to see if the machine is already booted and only attempts to boot the machine if neccessary.

If `only_if_needed` is False, the boot commands will be sent to the target machine without checking if it is already booted or not.

Warning: If the machine has already been booted, sending the boot commands again will not 'reboot' the machine with the newly supplied boot image, even if `only_if_needed` is False.

check_booted [bool] If `check_booted` is True this method waits for the machine to be fully booted before returning. If False, this check is skipped and the function returns as soon as the machine's Ethernet interface is likely to be up (but not necessarily before booting has completed).

sv_overrides [{name: value, ...}] Additional arguments used to override the default values in the 'sv' struct defined in the struct file.

Returns

bool Returns True if the machine was sent boot commands, False if the machine was already booted.

Raises

rig.machine_control.machine_controller.SpiNNakerBootError Raised when `check_booted` is True and the boot process was unable to boot the machine. Also raised when `only_if_needed` is True and the remote host is a BMP.

Notes

The constants *rig.machine_control.boot.spinX_boot_options* can be used to specify boot parameters, for example:

```
controller.boot(**spin3_boot_options)
```

This is neccessary on boards such as SpiNN-3 boards if the more than LED 0 are required by an application since by default, only LED 0 is enabled.

discover_connections(*x=255*, *y=255*)
Attempt to discover all available Ethernet connections to a machine.

After calling this method, *MachineController* will attempt to communicate via the Ethernet connection on the same board as the destination chip for all commands.

If called multiple times, existing connections will be retained in preference to new ones.

Note: The system must be booted for this command to succeed.

Note: Currently, only systems comprised of multiple Ethernet-connected SpiNN-5 boards are supported.

Parameters

x [int]

y [int] (Optional) The coordinates of the chip to initially use to query the system for the set of live chips.

Returns

int The number of new connections established.

application(*app_id*)
Update the context to use the given application ID and stop the application when done.

For example:

```
with cn.application(54):
    # All commands in this block will use app_id=54.
    # On leaving the block `cn.send_signal("stop", 54)` is
    # automatically called.
```

get_software_version(*x=255*, *y=255*, *processor=0*)
Get the software version for a given SpiNNaker core.

Returns

:py:class:‘.CoreInfo‘ Information about the software running on a core.

get_ip_address(*x*, *y*)
Get the IP address of a particular SpiNNaker chip’s Ethernet link.

Returns

str or None The IPv4 address (as a string) of the chip's Ethernet link or None if the chip does not have an Ethernet connection or the link is currently down.

write(*address*, *data*, *x*, *y*, *p=0*)
Write a bytestring to an address in memory.

It is strongly encouraged to only read and write to blocks of memory allocated using *sdram_alloc()*. Additionally, *sdram_alloc_as_filelike()* can be used to safely wrap read/write access to memory with a file-like interface and prevent accidental access to areas outside the allocated block.

Parameters

address [int] The address at which to start writing the data. Addresses are given within the address space of a SpiNNaker core. See the SpiNNaker datasheet for more information.

data [bytes] Data to write into memory. Writes are automatically broken into a sequence of SCP write commands.

read(*address*, *length_bytes*, *x*, *y*, *p=0*)
Read a bytestring from an address in memory.

Parameters

address [int] The address at which to start reading the data.

length_bytes [int] The number of bytes to read from memory. Large reads are transparently broken into multiple SCP read commands.

Returns

:py:class:'bytes' The data is read back from memory as a bytestring.

write_across_link(*address*, *data*, *x*, *y*, *link*)
Write a bytestring to an address in memory on a neigbouring chip.

> **Warning:** This function is intended for low-level debug use only and is not optimised for performance nor intended for more general use.

This method instructs a monitor processor to send 'POKE' nearest-neighbour packets to a neighbouring chip. These packets are handled directly by the SpiNNaker router in the neighbouring chip, potentially allowing advanced debug or recovery of a chip rendered otherwise unreachable.

Parameters

address [int] The address at which to start writing the data. Only addresses in the system-wide address map may be accessed. Addresses must be word aligned.

data [bytes] Data to write into memory. Must be a whole number of words in length. Large writes are automatically broken into a sequence of SCP link-write commands.

x [int]

y [int] The coordinates of the chip from which the command will be sent, *not* the coordinates of the chip on which the write will be performed.

link [*rig.links.Links*] The link down which the write should be sent.

read_across_link(*address*, *length_bytes*, *x*, *y*, *link*)
Read a bytestring from an address in memory on a neigbouring chip.

> **Warning:** This function is intended for low-level debug use only and is not optimised for performance nor intended for more general use.

This method instructs a monitor processor to send 'PEEK' nearest-neighbour packets to a neighbouring chip. These packets are handled directly by the SpiNNaker router in the neighbouring chip, potentially allowing advanced debug or recovery of a chip rendered otherwise unreachable.

Parameters

address [int] The address at which to start reading the data. Only addresses in the system-wide address map may be accessed. Addresses must be word aligned.

length_bytes [int] The number of bytes to read from memory. Must be a multiple of four (i.e. a whole number of words). Large reads are transparently broken into multiple SCP link-read commands.

x [int]

y [int] The coordinates of the chip from which the command will be sent, *not* the coordinates of the chip on which the read will be performed.

link [*rig.links.Links*] The link down which the read should be sent.

Returns

:py:class:'bytes' The data is read back from memory as a bytestring.

read_struct_field(*struct_name*, *field_name*, *x*, *y*, *p=0*)

Read the value out of a struct maintained by SARK.

This method is particularly useful for reading fields from the `sv` struct which, for example, holds information about system status. See `sark.h` for details.

Parameters

struct_name [string] Name of the struct to read from, e.g., *"sv"*

field_name [string] Name of the field to read, e.g., *"eth_addr"*

Returns

value The value returned is unpacked given the struct specification.

Currently arrays are returned as tuples, e.g.:

```
# Returns a 20-tuple.
cn.read_struct_field("sv", "status_map")

# Fails
cn.read_struct_field("sv", "status_map[1]")
```

write_struct_field(*struct_name*, *field_name*, *values*, *x*, *y*, *p=0*)

Write a value into a struct.

This method is particularly useful for writing values into the `sv` struct which contains some configuration data. See `sark.h` for details.

Parameters

struct_name [string] Name of the struct to write to, e.g., *"sv"*

field_name [string] Name of the field to write, e.g., *"random"*

values : Value(s) to be written into the field.

.. warning:: Fields which are arrays must currently be written in their entirety.

read_vcpu_struct_field(*field_name*, *x*, *y*, *p*)
Read a value out of the VCPU struct for a specific core.

Similar to *read_struct_field()* except this method accesses the individual VCPU struct for to each core and contains application runtime status.

Parameters

field_name [string] Name of the field to read from the struct (e.g. *"cpu_state"*)

Returns

value A value of the type contained in the specified struct field.

write_vcpu_struct_field(*field_name*, *value*, *x*, *y*, *p*)
Write a value to the VCPU struct for a specific core.

Parameters

field_name [string] Name of the field to write (e.g. *"user0"*)

value : Value to write to this field.

get_processor_status(*p*, *x*, *y*)
Get the status of a given core and the application executing on it.

Returns

:py:class:'.ProcessorStatus' Representation of the current state of the processor.

get_iobuf(*p*, *x*, *y*)
Read the messages `io_printf`'d into the `IOBUF` buffer on a specified core.

See also: *get_iobuf_bytes()* which returns the undecoded raw bytes in the `IOBUF`. Useful if the IOBUF contains non-text or non-UTF-8 encoded text.

Returns

str The string in the `IOBUF`, decoded from UTF-8.

get_iobuf_bytes(*p*, *x*, *y*)
Read raw bytes `io_printf`'d into the `IOBUF` buffer on a specified core.

This may be useful when the data contained in the `IOBUF` is not UTF-8 encoded text.

See also: *get_iobuf()* which returns a decoded string rather than raw bytes.

Returns

bytes The raw, undecoded string data in the buffer.

get_router_diagnostics(*x*, *y*)
Get the values of the router diagnostic counters.

Returns

:py:class:'~.RouterDiagnostics' Description of the state of the counters.

iptag_set(*iptag*, *addr*, *port*, *x*, *y*)
Set the value of an IPTag.

Forward SDP packets with the specified IP tag sent by a SpiNNaker application to a given external IP address.

A *tutorial example* of the use of IP Tags to send and receive SDP packets to and from applications is also available.

Parameters

iptag [int] Index of the IPTag to set

addr [string] IP address or hostname that the IPTag should point at.

port [int] UDP port that the IPTag should direct packets to.

iptag_get(*iptag*, *x*, *y*)
Get the value of an IPTag.

Parameters

iptag [int] Index of the IPTag to get

Returns

:py:class:‘.IPTag‘ The IPTag returned from SpiNNaker.

iptag_clear(*iptag*, *x*, *y*)
Clear an IPTag.

Parameters

iptag [int] Index of the IPTag to clear.

set_led(*led*, *action=None*, *x=Required*, *y=Required*)
Set or toggle the state of an LED.

Note: By default, SARK takes control of LED 0 and so changes to this LED will not typically last long enough to be useful.

Parameters

led [int or iterable] Number of the LED or an iterable of LEDs to set the state of (0-3)

action [bool or None] State to set the LED to. True for on, False for off, None to toggle (default).

fill(*address*, *data*, *size*, *x*, *y*, *p*)
Fill a region of memory with the specified byte.

Parameters

data [int] Data with which to fill memory. If *address* and *size* are word aligned then *data* is assumed to be a word; otherwise it is assumed to be a byte.

Notes

If the address and size are word aligned then a fast fill method will be used, otherwise a much slower write will be incurred.

sdram_alloc(*size*, *tag=0*, *x=Required*, *y=Required*, *app_id=Required*, *clear=False*)
Allocate a region of SDRAM for an application.

Requests SARK to allocate a block of SDRAM for an application and raises a *SpiNNakerMemoryError* on failure. This allocation will be freed when the application is stopped.

Parameters

size [int] Number of bytes to attempt to allocate in SDRAM.

tag [int] 8-bit tag that can be looked up by a SpiNNaker application to discover the address of the allocated block. The tag must be unique for this `app_id` on this chip. Attempting to allocate two blocks on the same chip and for the same `app_id` will fail. If `0` (the default) then no tag is applied.

For example, if some SDRAM is allocated with `tag=12`, a SpiNNaker application can later discover the address using:

```
void *allocated_data = sark_tag_ptr(12, 0);
```

A common convention is to allocate one block of SDRAM per application core and give each allocation the associated core number as its tag. This way the underlying SpiNNaker applications can simply call:

```
void *allocated_data = sark_tag_ptr(sark_core_id(), 0);
```

clear [bool] If True the requested memory will be filled with zeros before the pointer is returned. If False (the default) the memory will be left as-is.

Returns

int Address of the start of the region.

The allocated SDRAM remains valid until either the 'stop' signal is sent to the application ID associated with the allocation or *sdram_free()* is called on the address returned.

Raises

rig.machine_control.machine_controller.SpiNNakerMemoryError If the memory cannot be allocated, the tag is already taken or it is invalid.

sdram_alloc_as_filelike(*size*, *tag=0*, *x=Required*, *y=Required*, *app_id=Required*, *clear=False*)

Like *sdram_alloc()* but returns a *file-like object* which allows safe reading and writing to the block that is allocated.

Returns

:py:class:'.MemoryIO' File-like object which allows accessing the newly allocated region of memory. For example:

```
>>> # Read, write and seek through the allocated memory just
>>> # like a file
>>> mem = mc.sdram_alloc_as_filelike(12)
>>> mem.write(b"Hello, world")
12
>>> mem.seek(0)
>>> mem.read(5)
b"Hello"
>>> mem.read(7)
b", world"

>>> # Reads and writes are truncated to the allocated region,
>>> # preventing accidental clobbering/access of memory.
>>> mem.seek(0)
>>> mem.write(b"How are you today?")
12
>>> mem.seek(0)
>>> mem.read(100)
b"How are you "
```

See the *MemoryIO* class for details of other features of these file-like views of SpiNNaker's memory.

Raises

rig.machine_control.machine_controller.SpiNNakerMemoryError If the memory cannot be allocated, or the tag is already taken or invalid.

sdram_free(*ptr*, *x=Required*, *y=Required*)
Free an allocated block of memory in SDRAM.

Note: All unfreed SDRAM allocations associated with an application are automatically freed when the 'stop' signal is sent (e.g. after leaving a *application()* block). As such, this method is only useful when specific blocks are to be freed while retaining others.

Parameters

ptr [int] Address of the block of memory to free.

flood_fill_aplx(**args*, *app_id=Required*, *wait=True*, ***kwargs*)
Unreliably flood-fill APLX to a set of application cores.

Note: Most users should use the *load_application()* wrapper around this method which guarantees successful loading.

This method can be called in either of the following ways:

```
flood_fill_aplx("/path/to/app.aplx", {(x, y): {core, ...}, ...})
flood_fill_aplx({"/path/to/app.aplx": {(x, y): {core, ...}, ...},
                 ...})
```

Note that the latter format is the same format produced by `build_application_map()`.

Warning: The loading process is likely, but not guaranteed, to succeed. This is because the flood-fill packets used during loading are not guaranteed to arrive. The effect is that some chips may not receive the complete application binary and will silently ignore the application loading request.

As a result, the user is responsible for checking that each core was successfully loaded with the correct binary. At present, the two recommended approaches to this are:

- If the `wait` argument is given then the user should check that the correct number of application binaries reach the initial barrier (i.e., the `wait` state). If the number does not match the expected number of loaded cores the next approach must be used:
- The user can check the process list of each chip to ensure the application was loaded into the correct set of cores. See *read_vcpu_struct_field()*.

Parameters

app_id [int]

wait [bool (Default: True)] Should the application await the AppSignal.start signal after it has been loaded?

load_application(**args*, *n_tries=2*, *app_start_delay=0.1*, *app_id=Required*, *wait=False*, ***kwargs*)

Load an application to a set of application cores.

This method guarantees that once it returns, all required cores will have been loaded. If this is not possible after a small number of attempts, a *SpiNNakerLoadingError* will be raised.

This method can be called in either of the following ways:

```
load_application("/path/to/app.aplx", {(x, y): {core, ...}, ...})
load_application({"/path/to/app.aplx": {(x, y): {core, ...}, ...},
                  ...})
```

Note that the latter format is the same format produced by `build_application_map()`.

Parameters

app_id [int]

wait [bool] Leave the application in a wait state after successfully loading it.

n_tries [int] Number attempts to make to load the application.

app_start_delay [float] Time to pause (in seconds) after loading to ensure that the application successfully reaches the wait state before checking for success.

use_count [bool] If True (the default) then the targets dictionary will be assumed to represent _all_ the cores that will be loaded and a faster method to determine whether all applications have been loaded correctly will be used. If False a fallback method will be used.

Raises

rig.machine_control.machine_controller.SpiNNakerLoadingError This exception is raised after some cores failed to load after `n_tries` attempts.

send_signal(*signal*, *app_id*)

Transmit a signal to applications.

> **Warning:** In current implementations of SARK, signals are highly likely to arrive but this is not guaranteed (especially when the system's network is heavily utilised). Users should treat this mechanism with caution. Future versions of SARK may resolve this issue.

Parameters

signal [string or *AppSignal*] Signal to transmit. This may be either an entry of the *AppSignal* enum or, for convenience, the name of a signal (defined in *AppSignal*) as a string.

count_cores_in_state(*state*, *app_id*)

Count the number of cores in a given state.

> **Warning:** In current implementations of SARK, signals (which are used to determine the state of cores) are highly likely to arrive but this is not guaranteed (especially when the system's network is heavily utilised). Users should treat this mechanism with caution. Future versions of SARK may resolve this issue.

Parameters

state [string or *AppState* or]

iterable

Count the number of cores currently in this state. This may be either an entry of the *AppState* enum or, for convenience, the name of a state (defined in *AppState*) as a string or an iterable of these, in which case the total count will be returned.

wait_for_cores_to_reach_state(*state*, *count*, *app_id*, *poll_interval=0.1*, *timeout=None*)

Block until the specified number of cores reach the specified state.

This is a simple utility-wrapper around the *count_cores_in_state()* method which polls the machine until (at least) the supplied number of cores has reached the specified state.

Warning: In current implementations of SARK, signals (which are used to determine the state of cores) are highly likely to arrive but this is not guaranteed (especially when the system's network is heavily utilised). As a result, in uncommon-but-possible circumstances, this function may never exit. Users should treat this function with caution. Future versions of SARK may resolve this issue.

Parameters

state [string or *AppState*] The state to wait for cores to enter. This may be either an entry of the *AppState* enum or, for convenience, the name of a state (defined in *AppState*) as a string.

count [int] The (minimum) number of cores reach the specified state before this method terminates.

poll_interval [float] Number of seconds between state counting requests sent to the machine.

timeout [float or Null] Maximum number of seconds which may elapse before giving up. If None, keep trying forever.

Returns

int The number of cores in the given state (which will be less than the number required if the method timed out).

load_routing_tables(*routing_tables*, *app_id*)

Allocate space for an load multicast routing tables.

The routing table entries will be removed automatically when the associated application is stopped.

Parameters

routing_tables [{(x, y): [*RoutingTableEntry* (...), ...], ...}] Map of chip co-ordinates to routing table entries, as produced, for example by *routing_tree_to_tables()* and *minimise_tables()*.

Raises

rig.machine_control.machine_controller.SpiNNakerRouterError If it is not possible to allocate sufficient routing table entries.

load_routing_table_entries(*entries*, *x*, *y*, *app_id*)

Allocate space for and load multicast routing table entries into the router of a SpiNNaker chip.

Note: This method only loads routing table entries for a single chip. Most users should use *load_routing_tables()* which loads routing tables to multiple chips.

Parameters

entries [[*RoutingTableEntry*, ...]] List of *rig.routing_table.RoutingTableEntry*s.

Raises

rig.machine_control.machine_controller.SpiNNakerRouterError If it is not possible to allocate sufficient routing table entries.

get_routing_table_entries(*x*, *y*)
Dump the multicast routing table of a given chip.

Returns

[(:py:class:'~rig.routing_table.RoutingTableEntry', app_id, core) or None, ...] Ordered list of routing table entries with app_ids and core numbers.

clear_routing_table_entries(*x*, *y*, *app_id*)
Clear the routing table entries associated with a given application.

get_p2p_routing_table(*x*, *y*)
Dump the contents of a chip's P2P routing table.

This method can be indirectly used to get a list of functioning chips.

Note: This method only returns the entries for chips within the bounds of the system. E.g. if booted with 8x8 only entries for these 8x8 chips will be returned.

Returns

{(x, y): :py:class:'~rig.machine_control.consts.P2PTableEntry', ...}

get_chip_info(*x*, *y*)
Get general information about the resources available on a chip.

Returns

:py:class:'.ChipInfo' A named tuple indicating the number of working cores, the states of all working cores, the set of working links and the size of the largest free block in SDRAM and SRAM.

get_working_links(*x*, *y*)
Return the set of links reported as working.

This command tests each of the links leaving a chip by sending a PEEK nearest-neighbour packet down each link to verify that the remote device is a SpiNNaker chip. If no reply is received via a given link or if the remote device is not a SpiNNaker chip, the link is reported as dead.

See also: *get_chip_info()*.

Returns

set([:py:class:'rig.links.Links', ...])

get_num_working_cores(*x*, *y*)
Return the number of working cores, including the monitor.

See also: *get_chip_info()*.

get_system_info(*x=255*, *y=255*)
Discover the integrity and resource availability of a whole SpiNNaker system.

This command performs *get_chip_info()* on all working chips in the system returning an enhanced dict (*SystemInfo*) containing a look-up from chip coordinate to *ChipInfo*. In addition to standard dictionary functionality, *SystemInfo* provides a number of convenience methods, which allow convenient iteration over various aspects of the information stored.

Note: This method replaces the deprecated *get_machine()* method. To build a *Machine* for place-and-route purposes, the *rig.place_and_route.utils.build_machine()* utility function may be used with *get_system_info()* like so:

```
>> from rig.place_and_route.utils import build_machine
>> sys_info = mc.get_system_info()
>> machine = build_machine(sys_info)
```

Parameters

x [int]

y [int] The coordinates of the chip from which system exploration should begin, by default (255, 255). Most users will not need to change these parameters.

Returns

:py:class:‘.SystemInfo‘ An enhanced dict object {(x, y): *ChipInfo*, . . . } with a number of utility methods for accessing higher-level system information.

get_machine(*x=255*, *y=255*, *default_num_cores=18*)
Deprecated. Probe the machine to discover which cores and links are working.

Warning: This method has been deprecated in favour of *get_system_info()* for getting information about the general resources available in a SpiNNaker machine. This method may be removed in the future.

To build a *Machine* for place-and-route purposes, the *rig.place_and_route.utils.build_machine()* utility function may be used with *get_system_info()* like so:

```
>> from rig.place_and_route import build_machine
>> sys_info = mc.get_system_info()
>> machine = build_machine(sys_info)
```

This method also historically used the size of the SDRAM and SRAM heaps to set the respective resource values in the *Machine*. *get_machine()* since changed to reporting the size of the largest free block in the SDRAM and SRAM heaps on each chip. Most applications should not be negatively impacted by this change.

Note: The chip (x, y) supplied is the one where the search for working chips begins. Selecting anything other than (255, 255), the default, may be useful when debugging very broken machines.

Parameters

default_num_cores [int] This argument is ignored.

Returns

:py:class:‘~rig.place_and_route.Machine‘ This Machine will include all cores reported as working by the system software with the following resources defined:

Cores Number of working cores on each chip (including the monitor core, any cores already running applications and idle cores).

SDRAM The size of the largest free block of SDRAM on the heap. This gives a conservative measure of how much SDRAM is free on a given chip (which will underestimate availability if the system's memory is highly fragmented.

SRAM The size of the largest free block of SRAM on the heap. This gives a conservative measure of how much SRAM is free on a given chip (which will underestimate availability if the system's memory is highly fragmented.

class rig.machine_control.machine_controller.**MemoryIO**(*machine_controller*, *x*, *y*, *start_address*, *end_address*)

A file-like view into a subspace of the memory-space of a chip.

A *MemoryIO* is sliceable to allow construction of new, more specific, file-like views of memory.

For example:

```
>>> # Read, write and seek through memory as if it was a file
>>> f = MemoryIO(mc, 0, 1, 0x67800000, 0x6780000c)
>>> f.write(b"Hello, world")
12
>>> f.seek(0)
>>> f.read()
b"Hello, world"

>>> # Slice the MemoryIO to produce a new MemoryIO which can only
>>> # access a subset of the memory.
>>> g = f[0:5]
>>> g.read()
b"Hello"
>>> g.seek(0)
>>> g.write(b"Howdy, partner!")
5
>>> f.seek(0)
>>> f.read()
b"Howdy, world"
```

free()
Free the memory referred to by the file-like, any subsequent operations on this file-like or slices of it will fail.

address
Get the current hardware memory address (indexed from 0x00000000).

close()
Flush and close the file-like.

flush()
Flush any buffered writes.

This must be called to ensure that all writes to SpiNNaker made using this file-like object (and its siblings, if any) are completed.

Note: This method is included only for compatibility reasons and does nothing. Writes are not currently buffered.

---

read(*n_bytes=-1*)
Read a number of bytes from the memory.

Note: Reads beyond the specified memory range will be truncated.

Note: Produces a *TruncationWarning* if fewer bytes are read than requested. These warnings can be converted into exceptions using `warnings.simplefilter()`:

```
>>> import warnings
>>> from rig.machine_control.machine_controller \
...     import TruncationWarning
>>> warnings.simplefilter('error', TruncationWarning)
```

Parameters

n_bytes [int] A number of bytes to read. If the number of bytes is negative or omitted then read all data until the end of memory region.

Returns

:py:class:‘bytes‘ Data read from SpiNNaker as a bytestring.

seek(*n_bytes*, *from_what=0*)
Seek to a new position in the memory region.

Parameters

n_bytes [int] Number of bytes to seek.

from_what [int] As in the Python standard: *0* seeks from the start of the memory region, *1* seeks from the current position and *2* seeks from the end of the memory region. For example:

```
mem.seek(-1, 2)  # Goes to the last byte in the region
mem.seek(-5, 1)  # Goes 5 bytes before that point
mem.seek(0)      # Returns to the start of the region
```

Note that *os.SEEK_END*, *os.SEEK_CUR* and *os.SEEK_SET* are also valid arguments.

tell()
Get the current offset in the memory region.

Returns

int Current offset (starting at 0).

write(*bytes*)
Write data to the memory.

Note: Writes beyond the specified memory range will be truncated and a *TruncationWarning* is produced. These warnings can be converted into exceptions using `warnings.simplefilter()`:

```
>>> import warnings
>>> from rig.machine_control.machine_controller \
...     import TruncationWarning
>>> warnings.simplefilter('error', TruncationWarning)
```

Parameters

bytes [bytes] Data to write to the memory as a bytestring.

Returns

int Number of bytes written.

A high level interface for controlling a SpiNNaker system.

class rig.machine_control.machine_controller.**CoreInfo**

Information returned about a core by sver.

Parameters

position [(x, y)] Logical location of the chip in the system.

physical_cpu [int] The physical ID of the core. (Not useful to most users).

virt_cpu [int] The virtual ID of the core. This is the number used by all high-level software APIs.

software_version [(major, minor, patch)] The numerical components of the software version number. See also: `software_version_labels`.

buffer_size [int] Maximum supported size (in bytes) of the data portion of an SCP packet.

build_date [int] The time at which the software was compiled as a unix timestamp. May be zero if not set.

version_string [string] Human readable, textual version information split in to two fields by a "/". In the first field is the kernal (e.g. SC&MP or SARK) and the second the hardware platform (e.g. SpiNNaker).

software_version_labels [string] Any additional labels or build information associated with the software version. (See also: `software_version` and the Semantic Versioning specification).

class rig.machine_control.machine_controller.**ChipInfo**

Information returned about a chip.

If some parameter is omitted from the constructor, realistic defaults are provided. These should only be used for writing tests and general applications should set all values based on reports from the SpiNNaker machine itself, e.g. using *get_chip_info()*.

Parameters

num_cores [int] The number of working cores on the chip.

core_states [[*AppState*, ...]] The state of each working core in the machine in a list `num_cores` in length.

working_links [set([*rig.links.Links*, ...])] The set of working links leaving that chip. For a link to be considered working, the link must work in both directions and the device at the far end must also be a SpiNNaker chip.

largest_free_sdram_block [int] The size (in bytes) of the largest free block of SDRAM.

largest_free_sram_block [int] The size (in bytes) of the largest free block of SRAM.

largest_free_rtr_mc_block [int] Number of entries in the largest free block of multicast router entries.

ethernet_up [bool] True if the chip's Ethernet connection is connected, False otherwise.

ip_address [str] The IP address of the Chip's Ethernet connection. If ethernet_up is False, the value of this field is unpredictable and should be ignored.

local_ethernet_chip [(x, y)] The coordinates of the 'nearest' Ethernet connected chip to this chip, corresponding with the value in `sv->eth_addr`.

Note: This value may not literally be the *nearest* Ethernet connected chip. For example, it could be the Ethernet connected chip on the same board as the chip or chosen by the system at boot by some process which evenly balances load.

---

class rig.machine_control.machine_controller.**SystemInfo**(*width*, *height*, **args*, ***kwargs*)

An enhanced `dict` containing a lookup from chip coordinates, (x, y), to chip information, *ChipInfo*.

This dictionary contains an entry for every working chip in a system and no entry for chips which are dead. In addition to normal dictionary functionality, a number of utility methods are provided for iterating over useful information, for example individual cores and links.

Attributes

width [int] The width of the system in chips.

height [int] The height of the system in chips.

__init__(*width*, *height*, **args*, ***kwargs*)

Construct a *SystemInfo* object.

Parameters

width [int] The width of the system, in chips.

height [int] The height of the system, in chips.

... Remaining arguments are passed directly to the `dict` constructor.

chips()

Iterate over the coordinates of working chips.

An alias for *__iter__()*, included for consistency.

Yields

(x, y) The coordinate of a working chip.

ethernet_connected_chips()

Iterate over the coordinates of Ethernet connected chips.

Yields

((x, y), str) The coordinate and IP address of each Ethernet connected chip in the system.

dead_chips()
Generate the coordinates of all dead chips.

Yields

(x, y) The coordinate of a dead chip.

links()
Generate the coordinates of all working links.

Yields

(x, y, :py:class:'rig.links.Links') A working link leaving a chip from the perspective of the chip. For example `(0, 0, Links.north)` would be the link going north from chip (0, 0) to chip (0, 1).

dead_links()
Generate the coordinates of all dead links leaving working chips.

Any link leading to a dead chip will also be included in the list of dead links. In non-torroidal SpiNNaker sysmtes (e.g. single SpiNN-5 boards), links on the periphery of the system will be marked as dead.

Yields

(x, y, :py:class:'rig.links.Links') A working link leaving a chip from the perspective of the chip. For example `(0, 0, Links.north)` would be the link going north from chip (0, 0) to chip (0, 1).

cores()
Generate the set of all cores in the system.

Yields

(x, y, p, :py:class:'~rig.machine_control.consts.AppState') A core in the machine, and its state. Cores related to a specific chip are yielded consecutively in ascending order of core number.

__contains__(*chip_core_or_link*)
Test if a given chip, core or link is present and alive.

Parameters

chip_core_or_link [tuple]

- If of the form (x, y, *Links*), checks the link is present.
- If of the form (x, y, p), checks the core is present.
- If of the form (x, y, p, *AppState*), checks the core is present and in the specified state.
- If of the form (x, y), checks the chip is present.

__weakref__
list of weak references to the object (if defined)

class rig.machine_control.machine_controller.**ProcessorStatus**
Information returned about the status of a processor.

Parameters

registers [list] Register values dumped during a runtime exception. (All zero by default.)

program_status_register [int] CPSR register (dumped during a runtime exception and zero by default).

stack_pointer [int] Stack pointer (dumped during a runtime exception and zero by default).

link_register [int] Link register (dumped during a runtime exception and zero by default).

rt_code [*RuntimeException*] Code for any run-time exception which may have occurred.

phys_cpu [int] The physical CPU ID.

cpu_state [*AppState*] Current state of the processor.

mbox_ap_msg [int]

mbox_mp_msg [int]

mbox_ap_cmd [int]

mbox_mp_cmd [int]

sw_count [int] Saturating count of software errors. (Calls to *sw_err*).

sw_file [int] Pointer to a string containing the file name in which the last software error occurred.

sw_line [int] Line number of the last software error.

time [int] Time application was loaded.

app_name [string] Name of the application loaded to the processor core.

iobuf_address [int] Address of the output buffer used by the processor.

app_id [int] ID of the application currently running on the processor.

version [(major, minor, patch)] The version number of the application running on the core.

user_vars [list] List of 4 integer values that may be set by the user.

class rig.machine_control.machine_controller.**RouterDiagnostics**

A namedtuple of values of a SpiNNaker router's 16 programmable diagnostic counters.

Counter values can be accessed by subscripting:

```
>>> diag = mc.get_router_diagnostics(0, 0)
>>> diag[0]
53491
```

On boot, the first twelve counters are preconfigured to count commonly used information. As a convenience, these counter values can be selected by name:

```
>>> diag.dropped_multicast
41
```

Note: It is possible to reconfigure *all* of the router counters to count arbitrary events (see the `rFN` register in section 10.11 of the SpiNNaker datasheet). If this has been done, using the subscript syntax for accessing counter values from this structure is strongly recommended.

Parameters

local_multicast [int]

external_multicast [int]

local_p2p [int]

external_p2p [int]

local_nearest_neighbour [int]

external_nearest_neighbour [int]

local_fixed_route [int]

external_fixed_route [int] For each of SpiNNaker's four packet types (multicast, point-to-point, nearest neighbour and fixed-route), there is:

- A `local_*` counter which reports the number of packets routed which were sent by local application cores.
- An `external_*` counter which reports the number of packets routed which were received from external sources (i.e. neighbouring chips).

Any packets which were dropped by the router are not included in these counts.

dropped_multicast [int]

dropped_p2p [int]

dropped_nearest_neighbour [int]

dropped_fixed_route [int] These counters report the number of each type of packet which were dropped after arrival at this core.

counter12 [int]

counter13 [int]

counter14 [int]

counter15 [int] These counters are disabled by default.

class rig.machine_control.machine_controller.**IPTag**

An IPTag as read from a SpiNNaker machine.

Parameters

addr [str] IP address SDP packets are forwarded to

mac [int]

port [int] Port number to forward SDP packets to

timeout [int]

count [int]

rx_port [int]

spinn_addr [int]

spinn_port [int]

exception rig.machine_control.machine_controller.**SpiNNakerBootError**

Raised when attempting to boot a SpiNNaker machine has failed.

__weakref__

list of weak references to the object (if defined)

exception rig.machine_control.machine_controller.**SpiNNakerMemoryError**(*size*, *x*, *y*, *tag=0*, *tag_in_use=False*)

Raised when it is not possible to allocate memory on a SpiNNaker chip.

Attributes

size [int] The size of the failed allocation.

chip [(x, y)] The chip coordinates on which the allocation failed.

tag [int] The tag number of the failed allocation.

tag_in_use [bool] Whether the allocation failed because the tag was already in use.

__init__(*size*, *x*, *y*, *tag=0*, *tag_in_use=False*)
x.__init__(...) initializes x; see help(type(x)) for signature

__str__(*)* <==> *str(x)*

__weakref__
list of weak references to the object (if defined)

exception rig.machine_control.machine_controller.**SpiNNakerRouterError**(*count*, *x*, *y*)
Raised when it is not possible to allocated routing table entries on a SpiNNaker chip.

Attributes

count [int] The number of routing table entries requested.

chip [(x, y)] The coordinates of the chip the allocation failed on.

__init__(*count*, *x*, *y*)
x.__init__(...) initializes x; see help(type(x)) for signature

__str__(*)* <==> *str(x)*

__weakref__
list of weak references to the object (if defined)

exception rig.machine_control.machine_controller.**SpiNNakerLoadingError**(*application_map*)
Raised when it has not been possible to load applications to cores.

Attributes

app_map [{"/path/to/app.aplx": {(x, y): {core, ...}, ...}, ...}] The application map of the cores which could not be loaded.

__init__(*application_map*)
x.__init__(...) initializes x; see help(type(x)) for signature

__str__(*)* <==> *str(x)*

__weakref__
list of weak references to the object (if defined)

exception rig.machine_control.machine_controller.**TruncationWarning**
Warning produced when a reading/writing past the end of a *MemoryIO* results in a truncated read/write.

__weakref__
list of weak references to the object (if defined)

rig.machine_control.utils.**sdram_alloc_for_vertices**(*controller*, *placements*, *allocations*, *core_as_tag=True*, *sdram_resource=SDRAM*, *cores_resource=Cores*, *clear=False*)
Allocate and return a file-like view of a region of SDRAM for each vertex which uses SDRAM as a resource.

The tag assigned to each region of assigned SDRAM is the index of the first core that each vertex is assigned. For example:

```
placements = {vertex: (0, 5)}
allocations = {vertex: {Cores: slice(3, 6),
                        SDRAM: slice(204, 304)}}
sdram_allocations = sdram_alloc_for_vertices(
    controller, placements, allocations
)
```

Will allocate a 100-byte block of SDRAM for the vertex which is allocated cores 3-5 on chip (0, 5). The region of SDRAM will be tagged *3* (because this is the index of the first core).

Parameters

controller [*rig.machine_control.MachineController*] Controller to use to allocate the SDRAM.

placements [{vertex: (x, y), ...}] Mapping of vertices to the chips they have been placed on. Same as produced by placers.

allocations [{vertex: {resource: allocation, ...}, ...}] Mapping of vertices to the resources they have been allocated.

A block of memory of the size specified by the *sdram_resource* (default: *SDRAM*) resource will be allocated for each vertex. Note that location of the supplied allocation is *not* used.

When *core_as_tag=True*, the tag allocated will be the ID of the first core used by the vertex (indicated by the *cores_resource*, default *Cores*), otherwise the tag will be set to 0.

clear [bool] If True the requested memory will be filled with zeros before the pointer is returned. If False (the default) the memory will be left as-is.

Returns

{vertex: :py:class:'.MemoryIO', ...} A file-like object for each vertex which can be used to read and write to the region of SDRAM allocated to the vertex.

Other Parameters

core_as_tag [bool] Use the index of the first allocated core as the tag for the region of memory, otherwise 0 will be used.

sdram_resource [resource (default *SDRAM*)] Key used to indicate SDRAM usage in the resources dictionary.

cores_resource [resource (default *Cores*)] Key used to indicate cores which have been allocated in the allocations dictionary.

Raises

rig.machine_control.machine_controller.SpiNNakerMemoryError If the memory cannot be allocated, or a tag is already taken or invalid.

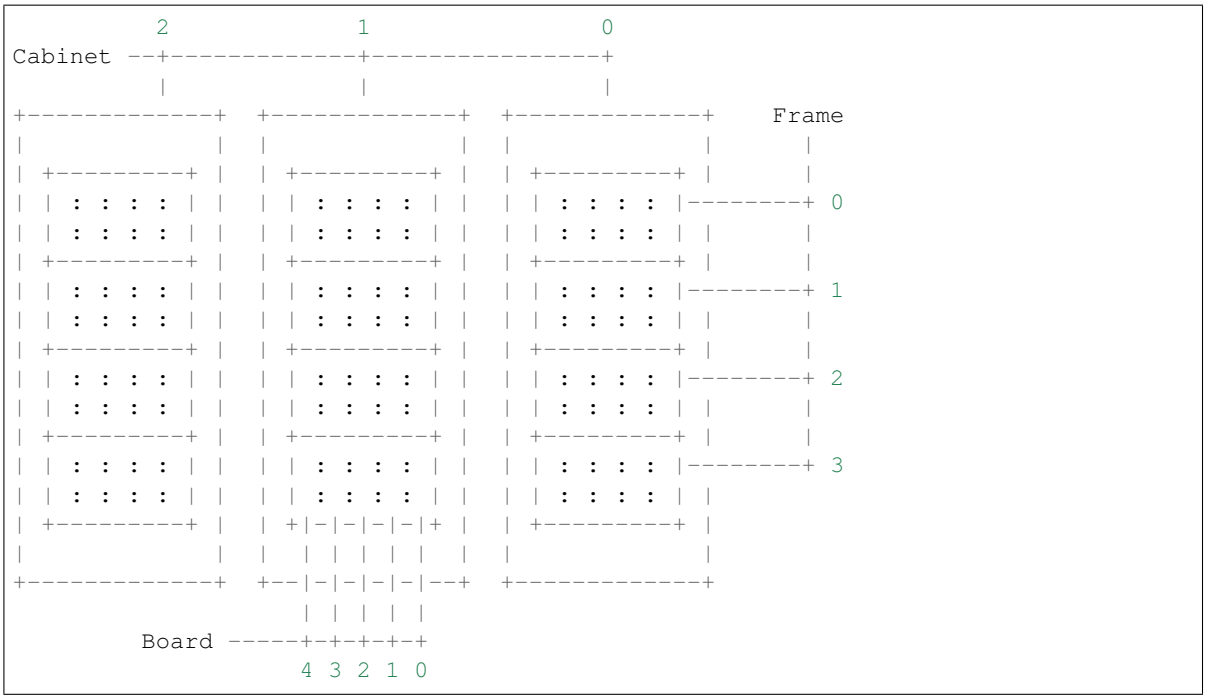
`BMPController`: BMP Control API

class rig.machine_control.**BMPController**(*hosts*, *scp_port=17893*, *n_tries=5*, *timeout=0.5*, *initial_context={'board': 0, 'cabinet': 0, 'frame': 0}*)

Control the BMPs (Board Management Processors) onboard SpiNN-5 boards in a SpiNNaker machine.

A *tutorial* is available which introduces the basic features of this class.

BMPs (and thus boards) are addressed as follows:



Coordinates are conventionally written as 3-tuples of integers (cabinet, frame, board). This gives the upper-right-most board's coordinate (0, 0, 0).

Communication with BMPs is facilitated either directly via Ethernet or indirectly via the Ethernet connection of another BMP and the CAN bus in the backplane of each frame.

This class aims not to be a complete BMP communication solution (users are referred instead to the general-purpose *bmpc* utility), but rather to cover common uses of the BMP in normal application usage.

__init__(*hosts*, *scp_port=17893*, *n_tries=5*, *timeout=0.5*, *initial_context={'board': 0, 'cabinet': 0, 'frame': 0}*)

Create a new controller for BMPs in a SpiNNaker machine.

Parameters

hosts [string or {coord: string, ...}] Hostname or IP address of the BMP to connect to or alternatively, multiple addresses can be given in a dictionary to allow control of many boards. *coord* may be given as ether (cabinet, frame) or (cabinet, frame, board) tuples. In the former case, the address will be used to communicate with all boards in the specified frame except those listed explicitly. If only a single hostname is supplied it is assumed to be for all boards in cabinet 0, frame 0.

scp_port [int] Port number to use for all SCP connections

n_tries [int] Number of SDP packet retransmission attempts.

timeout [float] SDP response timeout.

initial_context [*{argument: value}*] Dictionary of default arguments to pass to methods in this class. This defaults to selecting the coordinate (0, 0, 0) which is convenient in single-board systems.

__call__(***context_args*)

Create a new context for use with *with*.

send_scp(**args*, *frame=Required*, *board=Required*, *cabinet=Required*, ***kwargs*)

Transmit an SCP Packet to a specific board.

Automatically determines the appropriate connection to use.

See the arguments for *SCPConnection()* for details.

Parameters

cabinet [int]

frame [int]

board [int]

get_software_version(*cabinet*, *frame*, *board*)
Get the software version for a given BMP.

Returns

:py:class:'.BMPInfo' Information about the software running on a BMP.

set_power(*state*, *cabinet*, *frame*, *board*, *delay=0.0*, *post_power_on_delay=5.0*)
Control power to the SpiNNaker chips and FPGAs on a board.

Returns

state [bool] True for power on, False for power off.

board [int or iterable] Specifies the board to control the power of. This may also be an iterable of multiple boards (in the same frame). The command will actually be sent board 0, regardless of the set of boards specified.

delay [float] Number of seconds delay between power state changes of different boards.

post_power_on_delay [float] Number of seconds for this command to block once the power on command has been carried out. A short delay (default) is useful at this point since power-supplies and SpiNNaker chips may still be coming on line immediately after the power-on command is sent.

> **Warning:** If the set of boards to be powered-on does not include board 0, this timeout should be extended by 2-3 seconds. This is due to the fact that BMPs immediately acknowledge power-on commands to boards other than board 0 but wait for the FPGAs to be loaded before responding when board 0 is powered on.

set_led(*led*, *action=None*, *cabinet=Required*, *frame=Required*, *board=Required*)
Set or toggle the state of an LED.

Note: At the time of writing, LED 7 is only set by the BMP on start-up to indicate that the watchdog timer reset the board. After this point, the LED is available for use by applications.

Parameters

led [int or iterable] Number of the LED or an iterable of LEDs to set the state of (0-7)

action [bool or None] State to set the LED to. True for on, False for off, None to toggle (default).

board [int or iterable] Specifies the board to control the LEDs of. This may also be an iterable of multiple boards (in the same frame). The command will actually be sent to the first board in the iterable.

read_fpga_reg(*fpga_num*, *addr*, *cabinet*, *frame*, *board*)
Read the value of an FPGA (SPI) register.

See the SpI/O project's spinnaker_fpga design's README for a listing of FPGA registers. The SpI/O project can be found on GitHub at: https://github.com/SpiNNakerManchester/spio/

Parameters

fpga_num [int] FPGA number (0, 1 or 2) to communicate with.

addr [int] Register address to read to (will be rounded down to the nearest 32-bit word boundary).

Returns

int The 32-bit value at that address.

write_fpga_reg(*fpga_num*, *addr*, *value*, *cabinet*, *frame*, *board*)
Write the value of an FPGA (SPI) register.

See the SpI/O project's spinnaker_fpga design's README for a listing of FPGA registers. The SpI/O project can be found on GitHub at: https://github.com/SpiNNakerManchester/spio/

Parameters

fpga_num [int] FPGA number (0, 1 or 2) to communicate with.

addr [int] Register address to read or write to (will be rounded down to the nearest 32-bit word boundary).

value [int] A 32-bit int value to write to the register

read_adc(*cabinet*, *frame*, *board*)
Read ADC data from the BMP including voltages and temperature.

Returns

:py:class:‘.ADCInfo‘

class rig.machine_control.bmp_controller.**BMPInfo**
Information returned about a BMP by sver.

Parameters

code_block [int] The BMP, on power-up, will execute the first valid block in its flash storage. This value which indicates which 64 KB block was selected.

frame_id [int] An identifier programmed into the EEPROM of the backplane which uniquely identifies the frame the board is in. Note: This ID is not necessarily the same as a board's frame-coordinate.

can_id [int] ID of the board in the backplane CAN bus.

board_id [int] The position of the board in a frame. (This should correspond exactly with a board's board-coordinate.

version [(major, minor, patch)] Software version number. See also: `version_labels`.

buffer_size [int] Maximum supported size (in bytes) of the data portion of an SCP packet.

build_date [int] The time at which the software was compiled as a unix timestamp. May be zero if not set.

version_string [string] Human readable, textual version information split in to two fields by a "/". In the first field is the kernel (e.g. BC&MP) and the second the hardware platform (e.g. Spin5-BMP).

version_labels [string] Any additional labels or build information associated with the software version. (See also: `version` and the Semantic Versioning specification).

class rig.machine_control.bmp_controller.**ADCInfo**

ADC data returned by a BMP including voltages and temperature.

Parameters

voltage_1_2a [float] Measured voltage on the 1.2 V rail A.

voltage_1_2b [float] Measured voltage on the 1.2 V rail B.

voltage_1_2c [float] Measured voltage on the 1.2 V rail C.

voltage_1_8 [float] Measured voltage on the 1.8 V rail.

voltage_3_3 [float] Measured voltage on the 3.3 V rail.

voltage_supply [float] Measured voltage of the (12 V) power supply input.

temp_top [float] Temperature near the top of the board (degrees Celsius)

temp_btm [float] Temperature near the bottom of the board (degrees Celsius)

temp_ext_0 [float] Temperature read from external sensor 0 (degrees Celsius) or None if not connected.

temp_ext_1 [float] Temperature read from external sensor 1 (degrees Celsius) or None if not connected.

fan_0 [int] External fan speed (RPM) of fan 0 or None if not connected.

fan_1 [int] External fan speed (RPM) of fan 1 or None if not connected.

`boot`: Low-level Machine Booting API

Boot constructs for a SpiNNaker machine.

Warning: Implementation is reconstructed from a Perl implementation which forms a significant part of the documentation for this process.

rig.machine_control.boot.**spin1_boot_options** = {'hw_ver': 1, 'led0': 483588}

Boot options for *boot()* for SpiNN-1 boards.

rig.machine_control.boot.**spin2_boot_options** = {'hw_ver': 2, 'led0': 24835}

Boot options for *boot()* for SpiNN-2 boards.

rig.machine_control.boot.**spin3_boot_options** = {'hw_ver': 3, 'led0': 1282}

Boot options for *boot()* for SpiNN-3 boards.

rig.machine_control.boot.**spin4_boot_options** = {'hw_ver': 4, 'led0': 1}

Boot options for *boot()* for SpiNN-4 boards.

rig.machine_control.boot.**spin5_boot_options** = {'hw_ver': 5, 'led0': 1}

Boot options for *boot()* for SpiNN-5 boards.

rig.machine_control.boot.**boot**(*hostname*, *boot_port=54321*, *scamp_binary=None*, *sark_struct=None*, *boot_delay=0.05*, *post_boot_delay=2.0*, *sv_overrides={}*, ***kwargs*)

Boot a SpiNNaker machine of the given size.

Parameters

hostname [str] Hostname or IP address of the SpiNNaker chip to use to boot the system.

boot_port [int] The port number to sent boot packets to.

scamp_binary [filename or None] Filename of the binary to boot the machine with or None to use the SC&MP binary bundled with Rig.

sark_struct [filename or None] The 'sark.struct' file which defines the datastructures or None to use the one bundled with Rig.

boot_delay [float] Number of seconds to pause between sending boot data packets.

post_boot_delay [float] Number of seconds to wait after sending last piece of boot data to give SC&MP time to re-initialise the Ethernet interface. Note that this does *not* wait for the system to fully boot.

sv_overrides [{name: value, ...}] Values used to override the defaults in the 'sv' struct defined in the struct file.

Returns

{struct_name: :py:class:'~rig.machine_control.struct_file.Struct'} Layout of structs in memory.

Notes

The constants *rig.machine_control.boot.spinX_boot_options* provide boot parameters for specific SpiNNaker board revisions, for example:

```
boot("board1", **spin3_boot_options)
```

Will boot the Spin3 board connected with hostname "board1".

`rig.machine_control.unbooted_ping.`**`listen`**(*timeout=6.0*, *port=54321*)

Listen for a 'ping' broadcast message from an unbooted SpiNNaker board.

Unbooted SpiNNaker boards send out a UDP broadcast message every 4-ish seconds on port 54321. This function listens for such messages and reports the IP address that it came from.

Parameters

timeout [float] Number of seconds to wait for a message to arrive.

port [int] The port number to listen on.

Returns

str or None The IP address of the SpiNNaker board from which a ping was received or None if no ping was observed.

`packets`: Raw SDP/SCP Packet Packing/Unpacking

Representations of SDP and SCP Packets.

`class` `rig.machine_control.packets.`**`SDPPacket`**(*reply_expected=False*, *tag=255*, *dest_port=None*, *dest_cpu=None*, *src_port=7*, *src_cpu=31*, *dest_x=None*, *dest_y=None*, *src_x=0*, *src_y=0*, *data=''*)

An SDP Packet

__init__(*reply_expected=False*, *tag=255*, *dest_port=None*, *dest_cpu=None*, *src_port=7*, *src_cpu=31*, *dest_x=None*, *dest_y=None*, *src_x=0*, *src_y=0*, *data=''*)

Create a new SDPPacket.

Parameters

dest_x [int (0-255)] x co-ordinate of the chip to which the packet should be sent.

dest_y [int (0-255)] y co-ordinate of the chip to which the packet should be sent.

dest_cpu [int (0-17)] Index of the core which should receive the packet.

dest_port [int (0-7)] Port which should receive the packet (0 is reserved for debugging).

data [bytes] Data to append to the packet.

reply_expected [bool] True if a response to this packet is expected, if False (the default) no response is expected.

Other Parameters

tag [int] IPTag used to determine where to send packets over IPv4. The default (`0xff`) indicates a packet being transmitted into SpiNNaker.

src_port [int] Source port of the packet.

src_cpu [int] Source CPU of the packet.

src_x [int] Source x co-ordinate of the packet.

src_y [int] Source y co-ordinate of the packet.

.. note:: The default values for *tag*, *src_port*, *src_cpu*, *src_x* and *src_y* indicate a packet being transmitted to SpiNNaker over the network and will not require changing for this use.

classmethod from_bytestring(*bytestring*)

Create a new SDPPacket from a bytestring.

Returns

SDPPacket An SDPPacket containing the data from the bytestring.

bytestring

Convert the packet into a bytestring.

class rig.machine_control.packets.**SCPPacket**(*reply_expected=False*, *tag=255*, *dest_port=None*, *dest_cpu=None*, *src_port=7*, *src_cpu=31*, *dest_x=None*, *dest_y=None*, *src_x=0*, *src_y=0*, *cmd_rc=None*, *seq=0*, *arg1=None*, *arg2=None*, *arg3=None*, *data=''*)

An SCP Packet

__init__(*reply_expected=False*, *tag=255*, *dest_port=None*, *dest_cpu=None*, *src_port=7*, *src_cpu=31*, *dest_x=None*, *dest_y=None*, *src_x=0*, *src_y=0*, *cmd_rc=None*, *seq=0*, *arg1=None*, *arg2=None*, *arg3=None*, *data=''*)

Create a new SCP formatted packet.

Parameters

dest_x [int (0-255)] x co-ordinate of the chip to which the packet should be sent.

dest_y [int (0-255)] y co-ordinate of the chip to which the packet should be sent.

dest_cpu [int (0-17)] Index of the core which should receive the packet.

dest_port [int (0-7)] Port which should receive the packet (0 is reserved for debugging).

cmd_rc [int (1 word)] Command/return code of the packet. This will determine what action occurs if the packet is handled by SARK or SCAMP.

arg1 [int (1 word) or None] If None then ignored.

arg2 [int (1 word) or None] If None then ignored.

arg3 [int (1 word) or None] If None then ignored.

data [bytes] Data to append to the packet after *arg1*, *arg2* and *arg3*.

reply_expected [bool] True if a response to this packet is expected, if False (the default) no response is expected.

Other Parameters

tag [int] IPTag used to determine where to send packets over IPv4. The default (`0xff`) indicates a packet being transmitted into SpiNNaker.

src_port [int] Source port of the packet.

src_cpu [int] Source CPU of the packet.

src_x [int] Source x co-ordinate of the packet.

src_y [int] Source y co-ordinate of the packet.

seq [int] Sequence number of the packet, used when communicating the SCAMP or SARK.

.. note:: The default values for *tag*, *src_port*, *src_cpu*, *src_x* and *src_y* indicate a packet being transmitted to SpiNNaker over the network and will not require changing for this use.

classmethod from_bytestring(*scp_packet*, *n_args=3*)
Create a new SCPPacket from a bytestring.

Parameters

scp_packet [bytestring] Bytestring containing an SCP packet.

n_args [int] The number of arguments to unpack from the SCP data.

packed_data
Pack the data for the SCP packet.

__repr__()
Produce a human-readable summary of (the most important parts of) the packet.

`scp_connection`: High-performance SCP protocol implementation

This module presents a high-performance implementation of the SCP protocol which is used to communicate with SC&MP.

A blocking implementation of the SCP protocol.

class rig.machine_control.scp_connection.**scpcall**
Utility for specifying SCP packets which will be sent using *send_scp_burst()* and their callbacks.

..note:: The parameters are similar to the parameters for `send_scp` but for the addition of *callback* between *expected_args* and *timeout*.

Parameters

x [int]

y [int]

p [int]

cmd [int]

arg1 [int]

arg2 [int]

arg3 [int]

data [bytes]

callback [function] Function which will be called with the packet that acknowledges the transmission of this packet.

timeout [float] Additional timeout in seconds to wait for a reply on top of the default specified upon instantiation.

class rig.machine_control.scp_connection.**SCPConnection**(*spinnaker_host*, *port=17893*, *n_tries=5*, *timeout=0.5*)

Implements the SCP protocol for communicating with a SpiNNaker chip.

__init__(*spinnaker_host*, *port=17893*, *n_tries=5*, *timeout=0.5*)

Create a new communicator to handle control of the SpiNNaker chip with the supplied hostname.

Parameters

spinnaker_host [str] A IP address or hostname of the SpiNNaker chip to control.

port [int] Port number to send to.

n_tries [int] The maximum number of tries to communicate with the chip before failing.

timeout [float] The timeout to use on the socket.

send_scp(*buffer_size*, *x*, *y*, *p*, *cmd*, *arg1=0*, *arg2=0*, *arg3=0*, *data=''*, *expected_args=3*, *timeout=0.0*)

Transmit a packet to the SpiNNaker machine and block until an acknowledgement is received.

Parameters

buffer_size [int] Number of bytes held in an SCP buffer by SARK, determines how many bytes will be expected in a socket.

x [int]

y [int]

p [int]

cmd [int]

arg1 [int]

arg2 [int]

arg3 [int]

data [bytestring]

expected_args [int] The number of arguments (0-3) that are expected in the returned packet.

timeout [float] Additional timeout in seconds to wait for a reply on top of the default specified upon instantiation.

Returns

:py:class:‘~rig.machine_control.packets.SCPPacket‘ The packet that was received in acknowledgement of the transmitted packet.

send_scp_burst(*buffer_size*, *window_size*, *parameters_and_callbacks*)
Send a burst of SCP packets and call a callback for each returned packet.

Parameters

buffer_size [int] Number of bytes held in an SCP buffer by SARK, determines how many bytes will be expected in a socket.

window_size [int] Number of packets which can be awaiting replies from the SpiNNaker board.

parameters_and_callbacks: iterable of :py:class:‘.scpcall‘ Iterable of *scpcall* elements. These elements can specify a callback which will be called with the returned packet.

read(*buffer_size*, *window_size*, *x*, *y*, *p*, *address*, *length_bytes*)
Read a bytestring from an address in memory.

..note:: This method is included here to maintain API compatibility with an alternative implementation of SCP.

Parameters

buffer_size [int] Number of bytes held in an SCP buffer by SARK, determines how many bytes will be expected in a socket and how many bytes of data will be read back in each packet.

window_size [int]

x [int]

y [int]

p [int]

address [int] The address at which to start reading the data.

length_bytes [int] The number of bytes to read from memory. Large reads are transparently broken into multiple SCP read commands.

Returns

:py:class:‘bytes‘ The data is read back from memory as a bytestring.

write(*buffer_size*, *window_size*, *x*, *y*, *p*, *address*, *data*)
Write a bytestring to an address in memory.

..note:: This method is included here to maintain API compatibility with an alternative implementation of SCP.

Parameters

buffer_size [int] Number of bytes held in an SCP buffer by SARK, determines how many bytes will be expected in a socket and how many bytes will be written in each packet.

window_size [int]

x [int]

y [int]

p [int]

address [int] The address at which to start writing the data. Addresses are given within the address space of a SpiNNaker core. See the SpiNNaker datasheet for more information.

data [bytes] Data to write into memory. Writes are automatically broken into a sequence of SCP write commands.

close()
Close the SCP connection.

__weakref__
list of weak references to the object (if defined)

exception rig.machine_control.scp_connection.**SCPError**(*message=''*, *packet=None*)
Base Error for SCP return codes.

Attributes

packet [*rig.machine_control.packets.SCPPacket*] The packet being processed when the error occurred. May be None if no specific packet was involved.

__init__(*message=''*, *packet=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

__weakref__
list of weak references to the object (if defined)

exception rig.machine_control.scp_connection.**TimeoutError**(*message=''*, *packet=None*)
Raised when an SCP is not acknowledged within the given period of time.

exception rig.machine_control.scp_connection.**FatalReturnCodeError**(*return_code=None*, *packet=None*)
Raised when an SCP command returns with an error which is connsidered fatal.

Attributes

return_code [*rig.machine_control.consts.SCPReturnCodes* or int] The return code (will be a raw integer if the code is unrecognised).

__init__(*return_code=None*, *packet=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

struct_file: SC&MP Struct File Reading

Read struct files for SARK/SC&MP.

rig.machine_control.struct_file.**read_struct_file**(*struct_data*)
Interpret a struct file defining the location of variables in memory.

Parameters

struct_data [bytes] String of bytes containing data to interpret as the struct definition.

Returns

{struct_name: :py:class:'~.Struct'} A dictionary mapping the struct name to a *Struct* instance. **Note:** the struct name will be a string of bytes, e.g., *b"vcpu"*.

rig.machine_control.struct_file.**num**(*value*)
Convert a value from one of several bases to an int.

class rig.machine_control.struct_file.**Struct**(*name*, *size=None*, *base=None*)
Represents an instance of a struct.

Elements in the struct are accessible by name, e.g., *struct[b"link_up"]* and are of type *StructField*.

Attributes

name [str] Name of the struct.

size [int] Total size of the struct in bytes.

base [int] Base address of struct in memory.

fields [{field_name: *StructField*}] Fields of the struct.

__init__(*name*, *size=None*, *base=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

update_default_values(***updates*)
Replace the default values of specified fields.

Parameters

Parameters are taken as keyword-arguments of 'field=new_value'.

Raises

KeyError If a field doesn't exist in the struct.

__setitem__(*name*, *field*)
Set a field in the struct.

__getitem__(*name*)
Get a field in the struct.

pack()
Pack the struct (and its default values) into a string of bytes.

Returns

:py:class:'bytes' Byte-string representation of struct containing default values.

__weakref__
list of weak references to the object (if defined)

class rig.machine_control.struct_file.**StructField**

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *pack_chars*, *offset*, *printf*, *default*, *length*)
Create new instance of StructField(pack_chars, offset, printf, default, length)

__repr__()
Return a nicely formatted representation string

default
Alias for field number 3

length
Alias for field number 4

offset
Alias for field number 1

pack_chars
Alias for field number 0

printf
Alias for field number 2

consts: Machine and Protocol Constants

Constants used in the SCP protocol.

rig.machine_control.consts.**BOOT_PORT = 54321**
Port used to boot a SpiNNaker machine.

rig.machine_control.consts.**SCP_PORT = 17893**
Port used for SDP communication.

rig.machine_control.consts.**SDP_HEADER_LENGTH = 8**
The number of bytes making up the header of an SDP packet.

rig.machine_control.consts.**SCP_SVER_RECEIVE_LENGTH_MAX = 512**
The smallest power of two large enough to handle that SVER will produce (256 + 8 bytes).

rig.machine_control.consts.**SPINNAKER_RTR_BASE = 3774873600**
Base address of router hardware registers.

rig.machine_control.consts.**SPINNAKER_RTR_P2P = 3774939136**
Base address of P2P routing table.

rig.machine_control.consts.**BMP_POWER_ON_TIMEOUT = 5.0**
Additional timeout for BMP power-on commands to reply.

rig.machine_control.consts.**BMP_ADC_MAX = 4096**
The range of values the BMP's 12-bit ADCs can measure.

rig.machine_control.consts.**BMP_V_SCALE_2_5 = 0.0006103515625**
Multiplier to convert from ADC value to volts for lines less than 2.5 V.

rig.machine_control.consts.**BMP_V_SCALE_3_3 = 0.00091552734375**
Multiplier to convert from ADC value to volts for 3.3 V lines.

rig.machine_control.consts.**BMP_V_SCALE_12 = 0.003662109375**
Multiplier to convert from ADC value to volts for 12 V lines.

rig.machine_control.consts.**BMP_TEMP_SCALE = 0.00390625**
Multiplier to convert from temperature probe values to degrees Celsius.

rig.machine_control.consts.**BMP_MISSING_TEMP = -32768**
Temperature value returned when a probe is not connected.

rig.machine_control.consts.**BMP_MISSING_FAN = -1**
Fan speed value returned when a fan is absent.

rig.machine_control.consts.**RTR_ENTRIES = 1024**
Number of routing table entries in each routing table.

rig.machine_control.consts.**RTE_PACK_STRING = '<2H 3I'**
Packing string used with routing table entries, values are (next, free, route, key, mask).

class rig.machine_control.consts.**SCPCommands**
Command codes used in SCP packets.

Attributes

sver = 0

read = 2

write = 3

fill = 5

link_read = 17

link_write = 18

nearest_neighbour_packet = 20

signal = 22

flood_fill_data = 23

led = 25

iptag = 26

alloc_free = 28

router = 29

info = 31

bmp_info = 48

power = 57

sver = 0

Get the software version

class rig.machine_control.consts.**SCPReturnCodes**

SCP return codes

Attributes

ok = 128

len = 129

sum = 130

cmd = 131

arg = 132

port = 133

timeout = 134

route = 135

cpu = 136

dead = 137

buf = 138

p2p_noreply = 139

p2p_reject = 140

p2p_busy = 141

p2p_timeout = 142

pkt_tx = 143

rig.machine_control.consts.**RETRYABLE_SCP_RETURN_CODES = set([<SCPReturnCodes.sum: 130>, <S**
The set of *SCPReturnCodes* values which indicate a non-fatal retryable fault.

rig.machine_control.consts.**FATAL_SCP_RETURN_CODES = {<SCPReturnCodes.len: 129>: 'Bad comm**
The set of fatal SCP errors and a human-readable error.

class rig.machine_control.consts.**DataType**

Used to specify the size of data being read to/from a SpiNNaker machine over SCP.

Attributes

byte = 0

short = 1

word = 2

class rig.machine_control.consts.**LEDAction**
Indicate the action that should be applied to a given LED.

Attributes

toggle = 1

off = 2

on = 3

class rig.machine_control.consts.**IPTagCommands**
Indicate the action that should be performed to the given IPTag.

Attributes

set = 1

get = 2

clear = 3

class rig.machine_control.consts.**AllocOperations**
Used to allocate or free regions of SDRAM and routing table entries.

Attributes

alloc_sdram = 0

free_sdram_by_ptr = 1

free_sdram_by_tag = 2

alloc_rtr = 3

free_rtr_by_pos = 4

free_rtr_by_app = 5

class rig.machine_control.consts.**RouterOperations**
Operations that may be performed to the router.

Attributes

init = 0

clear = 1

load = 2

fixed_route_set_get = 3

class rig.machine_control.consts.**NNCommands**
Nearest Neighbour operations.

Attributes

flood_fill_start = 6

flood_fill_core_select = 7

flood_fill_end = 15

class rig.machine_control.consts.**NNConstants**
Constants for use with nearest neighbour commands.

Attributes

retry = 24

forward = 63

class rig.machine_control.consts.**AppFlags**
Flags for application loading.

Attributes

wait = 1

class rig.machine_control.consts.**AppState**
States that an application may be in.

Attributes

dead = 0

power_down = 1

runtime_exception = 2

watchdog = 3

init = 4

wait = 5

c_main = 6

run = 7

sync0 = 8

sync1 = 9

pause = 10

exit = 11

idle = 15

class rig.machine_control.consts.**RuntimeException**
Runtime exceptions as reported by SARK.

Attributes

none = 0

reset = 1

undefined_instruction = 2

svc = 3

prefetch_abort = 4

data_abort = 5

unhandled_irq = 6

unhandled_fiq = 7

unconfigured_vic = 8

abort = 9

malloc_failure = 10

division_by_zero = 11

event_startup_failure = 12

software_error = 13

iobuf_failure = 14

bad_enable = 15

null_pointer = 16

pkt_startup_failure = 17

timer_startup_failure = 18

api_startup_failure = 19

incompatible_version = 20

class rig.machine_control.consts.**AppSignal**

Signals that may be transmitted to applications.

Attributes

init = 0

power_down = 1

stop = 2

start = 3

sync0 = 4

sync1 = 5

pause = 6

cont = 7

exit = 8

timer = 9

usr0 = 10

usr1 = 11

usr2 = 12

usr3 = 13

class rig.machine_control.consts.**AppDiagnosticSignal**
Signals which interrogate the state of a machine.

Note that a value is returned when any of these signals is sent.

Attributes

OR = 0

AND = 1

count = 2

class rig.machine_control.consts.**MessageType**
Internally used to specify the type of a message.

Attributes

multicast = 0

peer_to_peer = 1

nearest_neighbour = 2

rig.machine_control.consts.**signal_types = {<AppSignal.init: 0>: <MessageType.nearest_neig**
Mapping from an *AppSignal* to the *MessageType* used to transmit it.

rig.machine_control.consts.**diagnostic_signal_types = {<AppDiagnosticSignal.OR: 0>: <Messag**
Mapping from an *AppDiagnosticSignal* to the *MessageType* used to transmit it.

class rig.machine_control.consts.**P2PTableEntry**
Routing table entry in the point-to-point SpiNNaker routing table.

Attributes

east = 0

north_east = 1

north = 2

west = 3

south_west = 4

south = 5

none = 6

monitor = 7

class rig.machine_control.consts.**BMPInfoType**
Type of information to return from a bmp_info SCP command.

Attributes

serial = 0

can_status = 2

adc = 3

ip_addr = 4

2.3.2 `rig.wizard`: Stock SpiNNaker-board info gathering wizards

Many applications require end-users to supply the details of a SpiNNaker system they wish to connect to. This module hopes to provide a more-friendly user interaction than just asking for an IP address in the general case.

This module contains a number of wizards which extract various pieces of information from a user by asking simple questions. A wizard is a generator function which generates sequences of questions to which the answers are fed back into the generator. For command-line applications, a wrapper script *cli_wrapper()* will do all the heavy-lifting:

```
>> from rig.wizard import (
..     ip_address_wizard, cli_wrapper)
>> resp = cli_wrapper(ip_address_wizard())
Would you like to auto-detect the SpiNNaker system's IP address?
    0: Auto-detect
    1: Manually Enter IP address or hostname
Select an option 0-1 (default: 0):

Make sure the SpiNNaker system is switched on and is not booted.
<Press enter to continue>

Discovering attached SpiNNaker systems...

>> resp
{'ip_address': '192.168.240.253'}
```

Third-parties whose needs are not met by the supplied CLI wizard interface are encouraged to build their own front-ends which support the wizard protocol. The wizard generator functions generate the following objects:

- *MultipleChoice* This tuple includes a question to be presented to the user along with a list of valid options to choose from and a default value to select (or None if no default exists). The generator should be sent the index of the user's selection.
- *Text* This tuple includes a question to be presented to the user. The generator should be sent the user's free-form text response as a string.
- *Prompt* This tuple indicates the user should be shown a message which they should read and acknowledge. No response is expected.
- *Info* This tuple indicates the user should be shown a message to which no response is required.

When the information has been collected successfully, *Success* is raised by the wizard with a *data* attribute containing a dictionary with the information gathered by the wizard. If the wizard fails, a *Failure* exception is thrown with a human-readable message.

class rig.wizard.**MultipleChoice**

default
Alias for field number 2

options
Alias for field number 1

question
Alias for field number 0

class rig.wizard.**Text**

question
Alias for field number 0

class rig.wizard.**Prompt**

message
Alias for field number 0

class rig.wizard.**Info**

message
Alias for field number 0

exception rig.wizard.**Failure**
Indicates that the wizard couldn't determine the information requested. The message indicates the reason.

exception rig.wizard.**Success**(*data*)
The wizard successfully gathered the information requested.

Attributes

data [dict] A dictionary containing the information requested.

rig.wizard.**dimensions_wizard**()
A wizard which attempts to determine the dimensions of a SpiNNaker system.

> **Warning:** Since SC&MP v2.0.0 it is not necessary to know the dimensions of a SpiNNaker machine in order to boot it. As a result, most applications will no longer require this wizard step.

Returns `{"dimensions": (x, y)}` via the *Success* exception.

rig.wizard.**ip_address_wizard**()
A wizard which attempts to determine the IP of a SpiNNaker system.

Returns `{"ip_address": "..."}` via the *Success* exception.

rig.wizard.**cat**(**wizards*)
A higher-order wizard which is the concatenation of a number of other wizards.

The resulting data is the union of all wizard outputs.

rig.wizard.**cli_wrapper**(*generator*)
Given a wizard, implements an interactive command-line human-friendly interface for it.

Parameters

generator A generator such as one created by calling `rig.wizard.wizard_generator()`.

Returns

dict or None Returns a dictionary containing the results of the wizard or None if the wizard failed.

2.4 Standalone utility applications

2.4.1 Standalone Utility Applications

The following command-line utility applications are installed alongside the Rig Python library. These utilities are simple wrappers around the Rig API with an emphasis on common system and debug tasks.

rig-boot

The `rig-boot` command lets you quickly and easily boot SpiNNaker systems from the command line:

```
$ rig-boot HOSTNAME
```

If you have one of the smaller SpiNN-1, SpiNN-2 or SpiNN-3 boards, the command above will result in only the green LED (LED0) being available. The `--spinN` arguments may be used when booting such machines to correctly configure the LEDs, e.g.:

```
$ rig-boot HOSTNAME --spin3
```

To get a complete listing of available options and supported SpiNNaker boards, type:

```
$ rig-boot --help
```

rig-power

The `rig-power` command lets you quickly and easily power on and off SpiNNaker systems consisting of SpiNN-5 boards via their Board Management Processors (BMP).

For example, to power cycle a SpiNN-5 board (or a 24-board frame thereof):

```
$ rig-power BMP_HOSTNAME
```

To power-off:

```
$ rig-power BMP_HOSTNAME off
```

To power-cycle board 3 and the last 12 boards in frame:

```
$ rig-power BMP_HOSTNAME -b 3,12-23
```

To get a complete listing of available options:

```
$ rig-power --help
```

rig-info

The `rig-info` command displays basic information about (booted) SpiNNaker systems and BMPs. The command accepts a single hostname as an argument and prints output such as the following:

```
$ rig-info SPINNAKER_BOARD_HOSTNAME
Device Type: SpiNNaker

Software: SC&MP v2.0.0 (Built 2016-03-17 08:13:18)

Machine dimensions: 8x8
Working chips: 48 (18 cores: 40, 17 cores: 8)
Network topology: mesh
Dead links: 0 (+ 48 to dead/missing cores)

Application states:
    scamp-133: 48 run
    sark: 808 idle
```

And for BMPs:

```
$ rig-info BMP_HOSTNAME
Device Type: BMP

Software: BC&MP v2.0.0 (Built 2016-03-16 14:42:38)
Code block in use: 1
Board ID (slot number): 0

1.2 V supply: 1.24 V, 1.24 V, 1.24 V
1.8 V supply: 1.81 V
3.3 V supply: 3.32 V
Input supply: 11.98 V

Temperature top: 28.9 *C
Temperature bottom: 30.0 *C
```

rig-discover

The `rig-discover` command listens for any attached unbooted SpiNNaker boards on the network. This can be used to determine the IP address of a locally attached board. Example:

```
$ rig-discover
192.168.240.253
```

If no machines are discovered, the command will exit after a short timeout without printing anything.

rig-iobuf

The `rig-iobuf` command prints the messages printed by an application's calls to `io_printf(IOBUF, ...)`. For example, printing the IOBUF for core 1 on chip 0, 0:

```
$ rig-iobuf HOSTNAME 0 0 1
Hello, world!
```

rig-ps

The `rig-ps` command enumerates every application running on a machine. For example:

```
$ rig-ps HOSTNAME
X   Y   P   State             Application      App ID
--- --- --- ----------------- ---------------- ------
  0   0   0 run               scamp-133             0
  0   0   1 sync0             network_tester       66
  0   0   2 sync0             network_tester       66
  0   0   3 sync0             network_tester       66
  0   0   4 sync0             network_tester       66
  0   0   5 sync0             network_tester       66
...snip...
```

The listing can be filtered by:

- Application ID with `--app-id` or `-a`
- Application name with `--name` or `-n`

- Application State with `--state` or `-s`

The above arguments accept regular expressions as their argument. These can be used, for example, to locate misbehaving application cores:

```
$ rig-ps HOSTNAME --state '(?!run)'
X   Y   P   State             Application      App ID
--- --- --- ----------------- ---------------- ------
  3   6  13 watchdog          network_tester       66
```

Finally, the listings can be carried out for just a particular chip or core by adding the optional 'x', 'y' and 'p' arguments (similar to the ybug 'ps' command):

```
$ rig-ps HOSTNAME 0 0 3
X   Y   P   State             Application      App ID
--- --- --- ----------------- ---------------- ------
  0   0   3 sync0             network_tester       66
```

rig-counters

The `rig-counters` command reads the router diagnostic counters for all chips in a SpiNNaker system and reports any changes in value. This can be useful, for example, when checking if (and where) an application is dropping packets.

In the simplest use case, simply call `rig-counters` with a SpiNNaker hostname as an argument, run your application and then press enter to see how many packets were dropped:

```
$ rig-counters HOSTNAME
time,dropped_multicast
<press enter>
8.7,234
```

In the example above, 234 packets were dropped. Note that the output is in the form of a CSV file. You can give the *–multiple'* option to allow multiple samples to be captured. In the example below we capture four samples:

```
$ rig-counters HOSTNAME --multiple > out.csv
<press enter>
<press enter>
<press enter>
<press enter>
<press enter> ^C
$ cat out.csv
time,dropped_multicast
1.0,12
1.4,34
2.3,23
2.7,11
```

Instead of manually pressing enter to trigger a sample, you can use the `--command` argument to report the number of dropped packets during the execution of your program:

```
$ rig-counters HOSTNAME --command ./my_program my_args
time,dropped_multicast
10.4,102
```

You can also report each router's counter values individually using the `--detailed` option:

```
$ rig-counters HOSTNAME --detailed
time,x,y,dropped_multicast
<press enter>
10.4,0,0,10
10.4,0,1,2
10.4,0,2,5
...
```

Other router counter values can be reported too, see `rig-counters --help` for more details.

Warning: `rig-counters` works by polling the router in every chip in a SpiNNaker machine. This process takes some time, is not atomic and also results in P2P messages being sent through the SpiNNaker network.

The system is polled once when the utility is started and then once more for each sample requested (e.g. every time you press enter). As a result, you should be careful to only start or trigger a poll when the machine is otherwise idle, for example, before or after your application runs.

CHAPTER 3

Indicies and Tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

H

I

L

S

T

U

W