Eigend Internals

Overview

Agents Are Not Plugins

An Agent is self contained.

Agents can communicate with each other in a variety of ways.

There is no Central Program.

EigenD is just a framework for running lots of Agents, optimising the communication between them. Although most Agents are written to be hosted by EigenD, they do not get anything other than the most primitive support from it.

Some Agents are completely standalone. Workbench, Browser and Commander are all separate programs.

Agent Communications

Indexes allow Agents to find each other.

State Trees allow Agents to query and track other Agents' Properties.

RPCs allow Agents to make changes and invoke operations.

Agents have a network address which is independent of the location of the agent. These addresses are enclosed in <>.

Indexes

Indexes allow Agents to locate each other.

The '<main>' index contains all Agents. There can be other indexes.

The 'bls' command dumps an Index.

```
/503/ bls '<main>'
<delay1>
<audio1>
<keygroup1>
<eigend1>
<console_mixer1>
<scale_manager1>
<interpreter1>
<rig1>
<pico_manager1>
<workbench>



/504/ bls '<language>'
<interpreter1>
```

State Trees

An Agent has a name and tree of simple State Variables. The network protocol allows another Agent to track this tree.

Each variable has an individual address '<audio1>#1.1'

The 'bcat' command shows the Tree.

```
/513/ bcat2 '<audio1>' -idDx
./rw {cname:audio,cordinal:1,protocols:,plugin:audio,cversion:1.0.0,timestamp:2,
     verbs: ... ,version:2.0.35-experimental}
1/rw/list {cname:inputs,protocols:create,timestamp:81132237023}
1.1/rw {domain:bfloat(-1,1,0,[]),cname:audio input,cordinal:1,
       protocols:input remove nostage,master:conn(None,None,'<console_mixer1>#1.1',None,None)}
1.2/rw {domain:bfloat(-1,1,0,[]),cname:audio input,cordinal:2,
       protocols:input remove nostage,master:conn(None,None,'<console_mixer1>#1.2',None,None)}
1.2.254/rw 0.0
2/rw/list {cname:outputs,protocols:create,timestamp:81132204606}
3/rw {domain:enumn(44100,48000,96000),master:,cname:sample rate,protocols:bind output set explicit input,
     verbs: ... }
```

Anatomy of a State Variable

Flags denote basic properties

- Read only vs Read write.
- Ignored by Setup Manager, or given special treatment.
- Whether fast data is present.

Slow Data – One value per variable

- Handled by slow threads. Generally handled in Python.
- Persistent - Stored by Setup Manager.

Fast Data – One Event per variable

- Optimised for fast handling. Not directly accessible in Python.
- Non persistent – Ignored by Setup Manager
- Fast Data carried in queues organised into Events.

Ports

Are the visible units of the system.

Are the things we see in Workbench and Stage, and the things we talk about in Belcanto.

May be composed of many State Variables.

Have names, meta data, persistent value, and performance data streams.

Anatomy of a Port

```
/517/ bcat '<audio1>#5'
5 {domain:bintn(0,64,0,[]),cname:output
channels,protocols:output}
5.254 0

/522/ bcat2 '<pico_manager1/1>#2' -idD
2 {domain:aniso([]),cname:pressure
output,protocols:output,slave:'<keygroup1>#12'} id=null
2.254  id=null
2.254.7  id=5 0.25
2.254.8  id=14 0.419921875
```

- A Port is a tree of State Variables
- The root is a dictionary of Metadata describing the Port.
- 254 sub tree contains data.
- 255 sub tree can contain anything.
- Other Children are child Ports.

Metadata

A dictionary which describes the Port.

- Name, Ordinal
- Data Type
- Protocols
- Verbs
- Connections
- And anything else...

You can add your own metadata to store whatever you need.

```
/530/ bcat '<scale_manager1>'

. {cname:scale manager,
   cordinal:1,
   ideals:note scale,
   protocols:,
   plugin:scale_manager,
   cversion:1.0.0,
   version:2.0.35-experimental}
   verbs:v(1,cancel([],None,option(None,[singular,numeric]))),,
         v(5,choose([],None,role(None,[ideal([None,scale]),singular]))),
         v(201,set([],~a,role(None,[partof(~s),notproto(set)]))),
         v(202,set([un],~a,role(None,[partof(~s),notproto(set)]))),
         v(203,up([],~a,role(None,[partof(~s),notproto(up)]))),
         v(204,down([],~a,role(None,[partof(~s),notproto(down)]))),
         v(205,set([toggle],~a,role(None,[partof(~s),notproto(set)]))),

1 {cname:note,protocols:virtual}

2 {cname:scale,protocols:browse virtual,timestamp:81132246950}

3 {cname:event,protocols:hidden-connection create}

252 {protocols:verb}
```

Protocols

Protocols flag any special behaviour the Port might implement.

A Protocol might imply anything about the tree rooted at this port.

- Special RPC's
- Particular Children
- Additional meta data
- Plumbing behaviour

```
/533/ bcat '<pico_manager1/1>#7'

7 {protocols:revconnect
             nostage
             input }


/534/ bcat '<pico_manager1/1>#249'

249 {protocols:input
               explicit
               output}
```

RPC

All configuration is done by RPC to a port.

There are standard RPC's to:

- Change Name
- Set Value
- Make connections
- Manage Lists
- Invoke Verbs
- Create Talker actions

RPC's are very simple. They take a string argument and return a string or an error string.

Useful RPC's

RPCs can be invoked from the command line.

The interpreter has some useful RPCs.

Dump
- Show connections

Identify
- Finds Address

Exec
- Runs Belcanto

```
/526/ brpc '<interpreter1>' dump pico keyboard 1
<main:pico_manager1/1>
keyboard pico 1 pressure output -> keygroup 1 pressure input
keyboard pico 1 controller output -> keygroup 1 controller input
keyboard pico 1 absolute strip output -> keygroup 1 absolute strip input 1
keyboard pico 1 absolute strip output -> keygroup 1 absolute strip input 2
keygroup 1 light output -> keyboard pico 1 light input
keyboard pico 1 activation output -> keygroup 1 activation input
keyboard pico 1 roll output -> keygroup 1 roll input
keyboard pico 1 key output -> keygroup 1 key input
keyboard pico 1 strip position output -> keygroup 1 strip position input 2
keyboard pico 1 strip position output -> keygroup 1 strip position input 1
keyboard pico 1 breath output -> keygroup 1 breath input
keyboard pico 1 yaw output -> keygroup 1 yaw input


/527/ brpc '<interpreter1>' identify pico keyboard 1
'<main:pico_manager1/1>' keyboard pico 1
```

Port Data

Ports can have:

- A single persistent value.
- Multiple fast performance outputs.

But not both. It's too confusing.

```
/543/ bcat -idD '<pico_manager1/1>#4.254'
4.254.1  id=4 -0.12060546875
4.254.2  id=15 0.0400390625
4.254.3  id=13 -0.2587890625
4.254.5  id=5 0.16943359375
4.254.6  id=16 0.1875
4.254.8  id=12 -0.1943359375
4.254.11  id=14 -0.06787109375
4.254.12  id=7 0.248046875


/544/ bcat -idD '<audio1>#5.254'
5.254 0 id=null
```

Policies

Every Port has a policy which defines how its data values behave.

Policies define the input and output behaviour.

Policies are in Python and you can make your own.

```
class Connector(atom.Atom):

    def __init__(self,controller,index,tag):
        atom.Atom.__init__(self,names="connector",ordinal=index,
            protocols='remove')

        self[1] = atom.Atom(
            domain=domain.BoundedInt(-32767,32767),
            names='key row',
            init=0,
            policy=atom.default_policy(self.__change_key_row),
            protocols="input explicit")

        self[4] = atom.Atom(
            domain=domain.Aniso(),
            names='output',
            policy=policy.FastReadOnlyPolicy(),
            protocols="connect-static output nostage")
```

Setup Management

The Setup Manager is part of EigenD but is in all respects a 'normal' Agent.

Setup Management uses only the State Variable and RPC mechanisms.

A Setup is loaded by sending a diff between the Agents current state and the state to be loaded.

State Variable flags affect this behaviour.

Setup Gotchas

State Variables in the setup which aren't in the Agent are normally ignored.

State Variables which are created as a side effect need to be marked as such so the Setup Manager knows to send them as part of the diff.

An example would be the effect send ports in a mixer channel, created as a side effect of creating the effect channel itself.

More Setup Gotchas

One needs to be clear about the distinction between creating an object as part of restoring a setup and creating it as a new object.

The initial creation will sometimes have side effects which must not be repeated on setup load, because they will be recreated automatically as well.

This is especially true with constructs which span different Agents, like Talkers or Connections.

Presave and Preload

The Setup manager only sends diffs, and only saves what is in the State Variables.

Presave and Preload RPC hooks allow Agents to make sure their state is correct for the diff.

Example: Audio Units only copy the plugin state into and out of the State Variables in these hooks because it's expensive.

Fast Data

Fast Data is optimised for use in the performance thread. It can't be directly manipulated in Python.

Fast Data is sent through data queues. There is usually quite a lot of it and it needs to handled efficiently.

Ports can have many channels of fast data. Each channel is the same kind of data, ie, key pressure.

Different types of data like key yaw and key roll are sent on their own Ports.

Events

Fast Data is organised into Events. Each Event has an ID, a beginning, and end, and a stream of data.

Each Port may transmit multiple Events simultaneously. Each must have an ID unique to that Port at that time. Different Ports can and will have events with the same ID at the same time.

This is how the Pressure, Yaw, and Roll signals for a particular key press are associated.

Event Example

Press keys 1 and 3 and look at the ports:

We have Events with IDs 1 and 3 on each of the pressure, roll, yaw, and activation Ports.

We have a single event '.' on the Strip.

```
/572/ bcat '<pico_manager1/1>'

2 {domain:aniso([]),cname:pressure output, … }
2.254.3  id=1 0.425537109375
2.254.4  id=3 0.229248046875

3 {domain:aniso([]),cname:roll output,... }
3.254.3  id=1 0.19677734375
3.254.4  id=3 -0.5322265625

4 {domain:aniso([]),cname:yaw output, … }
4.254.3  id=1 0.42724609375
4.254.4  id=3 -0.19677734375

5 {domain:aniso([]),cname:strip position output, … }
5.254  id=null
5.254.1  id=. -0.01156462542712688446044921875

8 {domain:aniso([]),cname:absolute strip output, … }
8.254  id=null
8.254.1  id=. 0.89183676242828369140625
```

Connections

Connections are made by configuring a Port to listen to the output of another Port.

The Port Data Policy then controls what happens to the incoming data.

Connections have a 'control' flag. Control connections are used to connect to the persistent value of a Port.

The connection is stored in the metadata for a Port.

```
/612/ bcat '<pico_manager1/1>#7'

7 {domain:aniso([]),
   cname:light input,
   protocols:revconnect nostage input,
   master:conn(None,None,'<keygroup1>#15',None,None)
   }




/613/ bcat2 '<keygroup1>#15'

15 {domain:aniso([]),
    cname:light output,
    protocols:revconnect output,
    slave:'<pico_manager1/1>#7'
    }



/614/ bcat '<delay1>#7.2.2.3'

7.2.2.3
  {domain:bool([]),
   cname:filter,
   protocols:input output,
   master:conn(None,None,'<interpreter1>#15.77',None,ctl)
  }
```

Bundles

Bundles carry events inside Agents. They form a processing graph.

Bundle Components written in C++ process or route Events.

Python creates and connects Components.

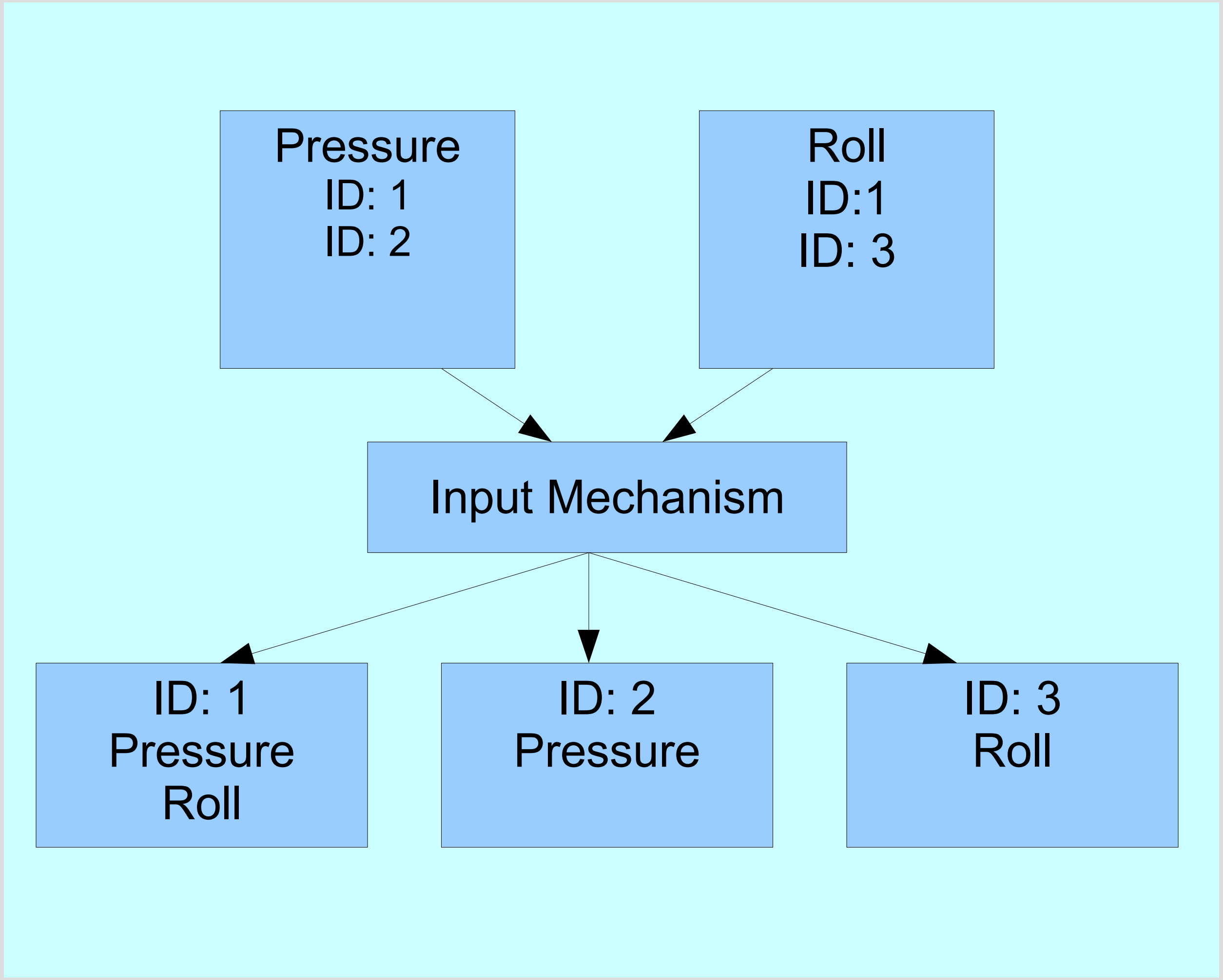
There are special input policies to get data into the bundle.

```
class Agent(agent.Agent):

  def __init__(self,address, ordinal):
    agent.Agent.__init__(self,names='sine oscillator', … )

    self[1] = bundles.Output(1,True,names="audio output")

    self.output = bundles.Splitter( … , self[1])
    self.osc = synth_native.sine(self.output.cookie(), … )
    self.input = bundles.VectorInput(self.osc.cookie(), … )

    self[2]=atom.Atom(domain=domain.BoundedFloat(0,1),
         names="volume input",
         policy=self.input.local_policy(1, … ))

    self[3]=atom.Atom(domain=domain.BoundedFloat(0,96000,rest=440),
         names="frequency input",
         policy=self.input.merge_policy(2,False))

   self[4]=atom.Atom(init=0.0, domain=domain.BoundedFloat(-1200,1200),
         names='detune input',
         policy=self.input.merge_policy(4,False))
```

Bundle Events

Inside Bundles, Events are organised by ID. Each Event may carry more than one signal.

This is the inverse of the external Port view, where one signal contains more than one Event.



Correlation

The inversion process is called 'Correlation' and uses the incoming Event ID's to match up signals that go together.

Exactly how this matching process works depends on the use of the incoming signals.

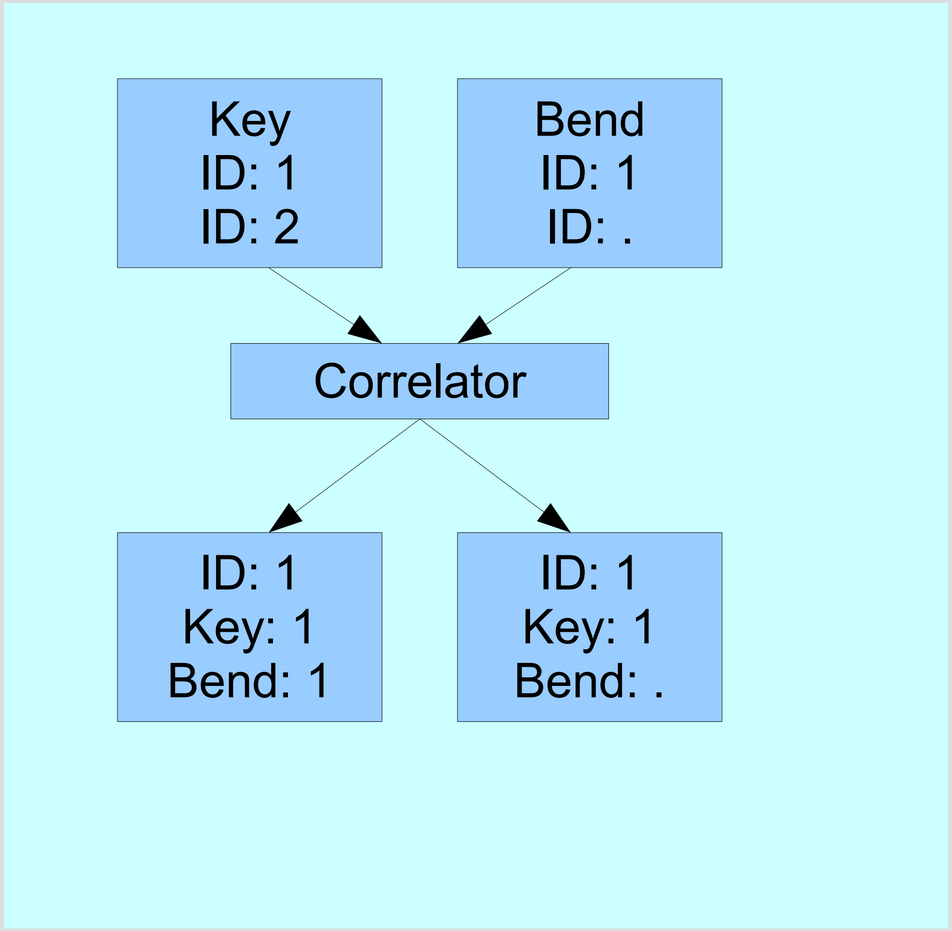
Some signals create Events inside an Agent, others merely provide additional data to existing Events.

An Scaler might have Key and Bend inputs. It can't do anything without an incoming Key Event. Key Events create frequency events on the output. Bend events affect these frequency Events, but are ignored without a corresponding Key Event.

Scaler Example

If the Bend is connected to Key Yaw, the Yaw will affect only the frequency of that Key, as the IDs of the Yaw and Key Events are the same.

If the Bend is connected to the Strip, the Strip will affect all Keys, as the Strip uses the empty ID '.' which prefixes all the key number events.

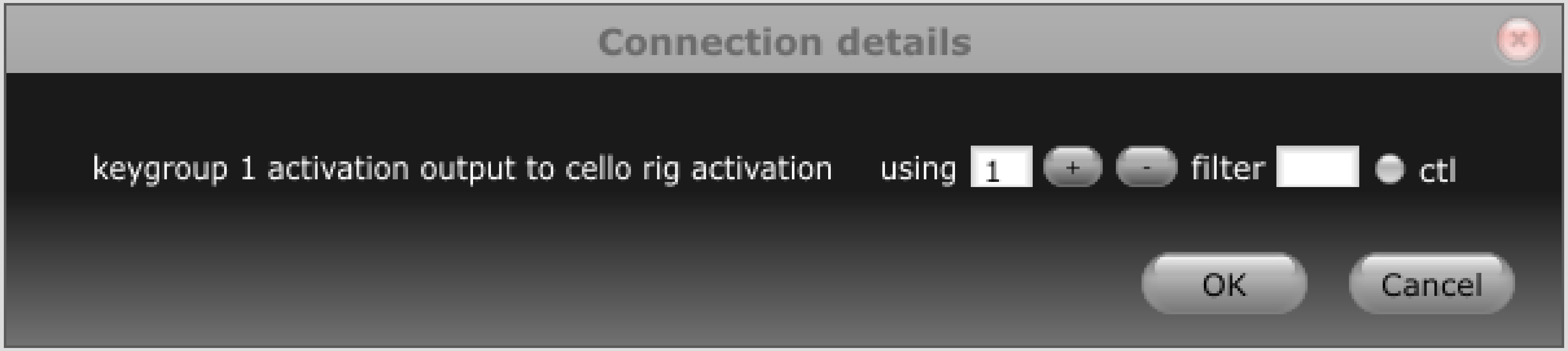


Transforming Event IDs

Each connection has two Event related parameters.

'Filter' causes only events prefixed by a particular value to be accepted. (The prefix is stripped off)

'Using' adds a leading value to incoming IDs



keygroup 1 activation output to cello rig activation using 1 filter

VectorInput

VectorInput is the primary class for performance input.

It creates a bundle, and can create Data Policies for Ports which dictate the role of the Port. (ie, Primary or Secondary)

You need to configure it with all the signals it will manage.

It acts as a factory for data policies.

```
class Agent(agent.Agent):

  def __init__(self, address, ordinal):
     agent.Agent.__init__(self, names='scaler', … )

     …

     self.output = bundles.Splitter(… ,self[1][1])
     self.filter = piw.scaler(self.output.cookie(),…)

     self.input = bundles.VectorInput(self.filter.cookie(),
          signal = (…,5,9) )

     self[4]=atom.Atom(names='inputs')

     self[4][22]=atom.Atom(
            domain=domain.Aniso(),
            policy=self.input.vector_policy(5,False),
            names='key input')

     self[4][8]=atom.Atom(
            domain=domain.BoundedFloat(-1,1),
            policy=self.input.merge_policy(9,False),
            names='key pitch bend input')
```

Iso and Aniso

Signals can be Anisochronous or Isochronous.

Aniso data is like MIDI data. It can be generated at any time and at any rate. Most of our signals are Aniso.

Audio data is Iso. It must be generated at a specific rate.

The input mechanism needs to know whether an given input is going to be used in an Iso or Aniso context. This is specified as an argument to the Data Policies constructed by VectorInput.

Conversion between the two types is automatic. You can, however, tailor the conversion process.

Other Policies

There are other, less used policies.

Normally, an an Event is started by a Primary input, and ends when there are no active Primary inputs.

Once the Event has ended, the bundle may still be active if the event needs to wind down (we call this lingering)

Ordinary secondary events no longer have any effect once the linger phase has begun. There are other policies which have different behaviour.

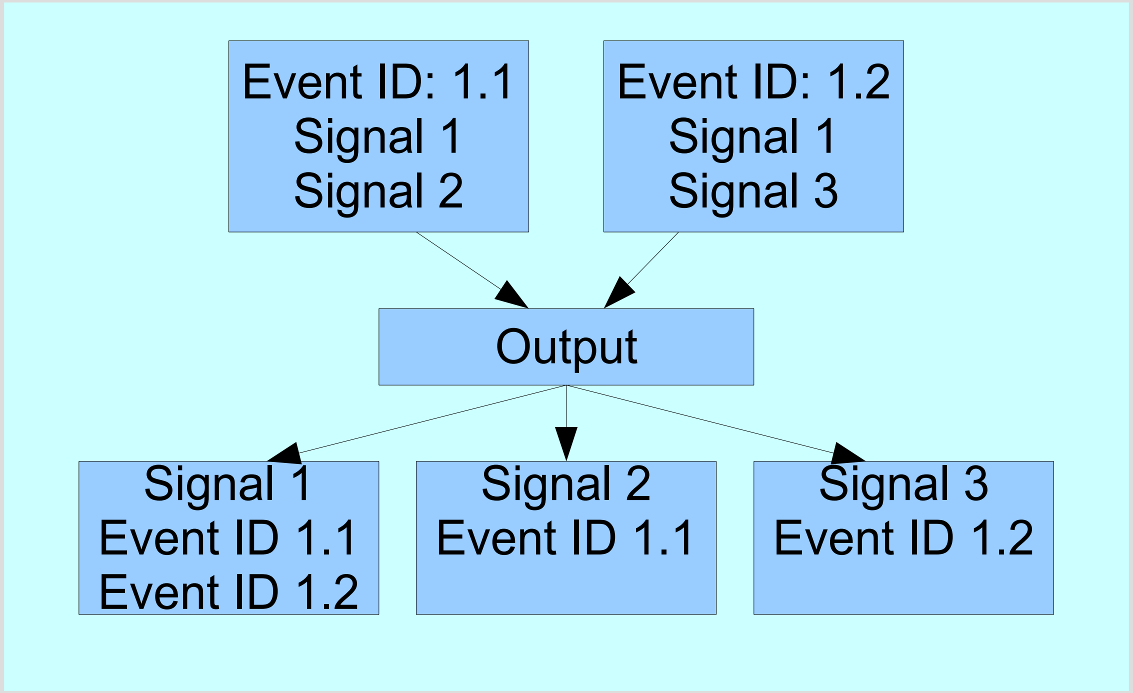
Think of a hold pedal.   This is a secondary Event which needs to prolong the Bundle Event.

Bundle Output

Bundle Outputs distribute the signals in each event onto the corresponding output ports.

This reverse correlation is generally straight forward.

```
class Agent(agent.Agent):

    def __init__(self,address, ordinal):
        agent.Agent.__init__(self,names='sine oscillator', … )

        self[1] = bundles.Output(1,True,names="audio output")

        self.output = bundles.Splitter(self[1], … )

        self.osc = synth_native.sine(self.output.cookie(), … )
```



Clocks

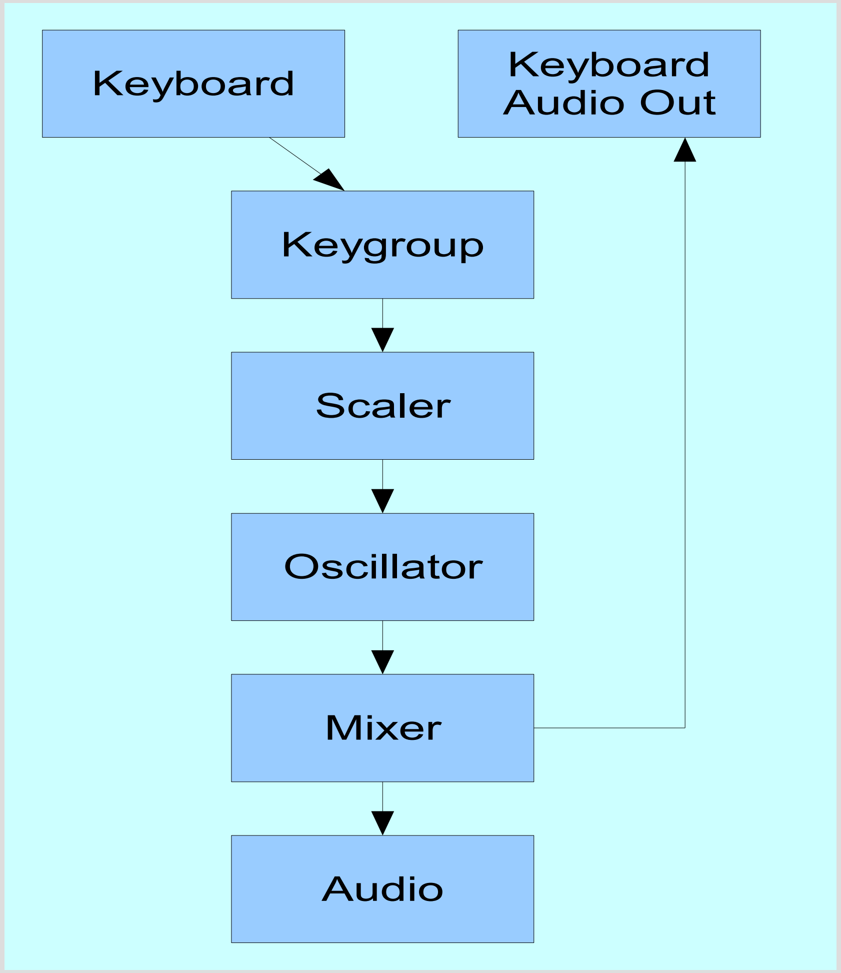
Data processing can be on demand, or batched into ticks.

The system provides clock nodes which provide tick callbacks for processing.

Ticks must be ordered correctly to avoid latency.

Each Port may have one clock. Each Bundle connection may have one clock.

Bundle Inputs deal with clock dependencies so that multiple inputs are handled correctly.



Clock Sources

Each clock node is associated with a clock source.

- Anisochronous Internal Clock.
  - This is used for non audio data. It ticks at a nominal 100hz.
- Isochronous Clock provided by Audio Agent.
  - Used for Audio data. The rate depends on the audio settings.
  - Iso outputs should generate one buffer per tick.

Iso clock nodes cannot be connected unless they have the same clock source. We plan to implement automatic resampling in the future.

And Finally

After stating that Agents are not plugins, of course they really are, in a technical sense.

As of release 2.0.35, EigenD ships with enough of an SDK (headers and a build system) to allow you to create and deploy Agents without requiring the full source from GitHub.

The way Agents are installed has also changed, to ease their exchange and packaging. Agents are contained in a single directory which can be freely copied between systems.

The Build system will also build you an installer which can be used to distribute an Agent.