Este guia é baseado na documentação para desenvolvedores LADSPA:
http://gdam.ffem.org/ladspa-doc/ladspa.html#toc1

Introdução ao LADSPA.

O que é LADSPA?

LADSPA vem de Linux Audio Development SIMPLE Plugin API, foi feito para ser uma forma simples de desenvolver plugins de áudio para Linux.

Instalando o LADSPA SDK.

http://www.ladspa.org/download/ladspa_sdk.tgz
baixe este pacote e digite em seu terminal:
$ tar -zxvf ladspa_sdk.tgz
$ cd ladspa_sdk
$ sudo make install

isso instalará o Ladspa SDK automaticamente em seu diretório /usr/local/bin/ e /usr/local/lib/ladspa/

configure o arquivo .bashrc localizado na sua pasta home para atualizar o LADSPA_PATH, adicione esta linha:

export LADSPA_PATH=/usr/local/lib/ladspa

Entendendo o Plugin Whitenoise da biblioteca de exemplos.

Tentarei descrever brevemente mostrando no código do whitenoise (que veio como exemplo na biblioteca LADSPA), o funcionamento básico de todo plugin.
Para começar, é necessário entender algumas terminologias comumente usadas na área:

Biblioteca de Plugins:
Uma biblioteca compartilhada que pode carregar ou descarregar a qualquer momento. Geralmente é carregada

Plugin:
Um único processor produtor/consumidor de sons, identificado por um descritor LADSPA_Descriptor. uma Biblioteca de Plugins pode conter diversos Plugins, cada um em um índice dentro da biblioteca. Os índices devem ser consecutivos e começar em 0.

Instância do Plugin:
É um gerador de unidade em execução. No host, geralmente é identificado por um “void* handle”.

Host:
É o programa que vai carregar os plugins. (Ardour, Audacity, etc…)

Os plugins LADSPA funcionam da seguinte maneira:

1 - A biblioteca de plugins é carregada.

2 - O descritor do plugin é obtido utilizando o tipo ladspa_descriptor da biblioteca, que deve alocar memória.
O descritor pode ser dividido em 2 partes:
Descrição dos dados:

```
g_psDescriptor->UniqueID
  = 1050;
g_psDescriptor->Label
  = strdup("noise_white");
g_psDescriptor->Properties
  = LADSPA_PROPERTY_HARD_RT_CAPABLE;
g_psDescriptor->Name
  = strdup("White Noise Source");
g_psDescriptor->Maker
  = strdup("Richard Furse (LADSPA example plugins)");
g_psDescriptor->Copyright
  = strdup("None");
g_psDescriptor->PortCount
  = 2;
piPortDescriptors
  = (LADSPA_PortDescriptor *)calloc(2, sizeof(LADSPA_PortDescriptor));
g_psDescriptor->PortDescriptors
  = (const LADSPA_PortDescriptor *)piPortDescriptors;
piPortDescriptors[NOISE_AMPLITUDE]
  = (LADSPA_PORT_INPUT
     | LADSPA_PORT_CONTROL);
piPortDescriptors[NOISE_OUTPUT]
  = LADSPA_PORT_OUTPUT | LADSPA_PORT_AUDIO;
pcPortNames
  = (char **)calloc(2, sizeof(char *));
g_psDescriptor->PortNames
  = (const char **)pcPortNames;
pcPortNames[NOISE_AMPLITUDE]
  = strdup("Amplitude");
pcPortNames[NOISE_OUTPUT]
  = strdup("Output");
psPortRangeHints = ((LADSPA_PortRangeHint *)
                    calloc(2, sizeof(LADSPA_PortRangeHint)));
g_psDescriptor->PortRangeHints
  = (const LADSPA_PortRangeHint *)psPortRangeHints;
psPortRangeHints[NOISE_AMPLITUDE].HintDescriptor
  = (LADSPA_HINT_BOUNDED_BELOW
     | LADSPA_HINT_LOGARITHMIC
     | LADSPA_HINT_DEFAULT_1);
psPortRangeHints[NOISE_AMPLITUDE].LowerBound
  = 0:
psPortRangeHints[NOISE_OUTPUT].HintDescriptor
  = 0;
```

Responsável pela alocação da memória e identificação do Plugin.
É interessante notar que o número e a identificação das portas já foi declarado, já sabemos que temos 2 portas, uma responsável pela amplitude do ruído e outra pelo output.
http://gdam.ffem.org/ladspa-doc/ladspa-2.html#ss2.2

E descrição da implementação:

```
    g_psDescriptor->instantiate
      = instantiateNoiseSource;
    g_psDescriptor->connect_port
      = NULL;
    g_psDescriptor->run
      = runNoiseSource;
    g_psDescriptor->run_adding
      = runAddingNoiseSource;
    g_psDescriptor->set_run_adding_gain
      = setNoiseSourceRunAddingGain;
    g_psDescriptor->deactivate
      = NULL;
    g_psDescriptor->cleanup
      = cleanupNoiseSource;
```

Resposável pela implementação do Plugin. É essa a ordem em que o plugin é executado.
http://gdam.ffem.org/ladspa-doc/ladspa-2.html#ss2.3

3 - O host usa a função instantiate do plugin para alocar um novo (ou várias novas) instâncias processadoras de samples.

```
/* Construct a new plugin instance. */
LADSPA_Handle
instantiateNoiseSource(const LADSPA_Descriptor * Descriptor,
                       unsigned long             SampleRate) {
  return malloc(sizeof(NoiseSource));
}
```

4 - O host deve conectar os buffers para cada uma das portas do plugin. Também deve chamar activate antes de enviar as samples pelo plugin.

```
/* Connect a port to a data location. */
void
connectPortToNoiseSource(LADSPA_Handle Instance,
                         unsigned long Port,
                         LADSPA_Data * DataLocation) {
  switch (Port) {
  case NOISE_AMPLITUDE:
    ((NoiseSource *)Instance)->m_pfAmplitudeValue = DataLocation;
    break;
  case NOISE_OUTPUT:
    ((NoiseSource *)Instance)->m_pfOutputBuffer = DataLocation;
    break;
  }
}
```

```
    g_psDescriptor->connect_port
      = connectPortToNoiseSource;
    g_psDescriptor->activate
      = NULL;
```

5 - O host processa os dados da sample com o plugin preenchendo os buffers de input que foram conectados, daí chamando o run ou run_adding. O host pode reconnectar portas com connect_port.

```
/* Run a delay line instance for a block of SampleCount samples. */
void
runNoiseSource(LADSPA_Handle Instance,
               unsigned long SampleCount) {

  NoiseSource * psNoiseSource;
  LADSPA_Data * pfOutput;
  LADSPA_Data fAmplitude;
  unsigned long lSampleIndex;

  psNoiseSource = (NoiseSource *)Instance;

  fAmplitude
    = *(psNoiseSource->m_pfAmplitudeValue) * (LADSPA_Data)(2.0 / RAND_MAX);

  pfOutput = psNoiseSource->m_pfOutputBuffer;
  for (lSampleIndex = 0; lSampleIndex < SampleCount; lSampleIndex++)
    *(pfOutput++) = (rand() - (RAND_MAX / 2)) * fAmplitude;
}
```

```c
/* Run a delay line instance for a block of SampleCount samples. *ADD*
   the output to the output buffer. */
void
runAddingNoiseSource(LADSPA_Handle Instance,
                     unsigned long SampleCount) {

  NoiseSource * psNoiseSource;
  LADSPA_Data * pfOutput;
  LADSPA_Data fAmplitude;
  unsigned long lSampleIndex;

  psNoiseSource = (NoiseSource *)Instance;

  fAmplitude
    = (*(psNoiseSource->m_pfAmplitudeValue)
       * psNoiseSource->m_fRunAddingGain
       * (LADSPA_Data)(2.0 / RAND_MAX));

  pfOutput = psNoiseSource->m_pfOutputBuffer;
  for (lSampleIndex = 0; lSampleIndex < SampleCount; lSampleIndex++)
    *(pfOutput++) += (rand() - (RAND_MAX / 2)) * fAmplitude;
}

void
setNoiseSourceRunAddingGain(LADSPA_Handle Instance,
                            LADSPA_Data   Gain) {
  ((NoiseSource *)Instance)->m_fRunAddingGain = Gain;
}
```

6 - O host desativa o handle do plugin. Você pode optar por ativar e reusar o handle, ou pode destruir o handle.

```c
    g_psDescriptor->deactivate
      = NULL;
```

Geralmente é deixado o valor NULL, ainda mais quando a função de cleanup vem logo depois.

7 - O handle é destruído usando a função cleanup.

```c
/* Throw away a simple delay line. */
void
cleanupNoiseSource(LADSPA_Handle Instance) {
  free(Instance);
}
```

8 - O plugin é fechado. Sua função _fini é a responsável por desalocar a memória.

```
/* _fini() is called automatically when the library is unloaded. */
void
_fini() {
  long lIndex;
  if (g_psDescriptor) {
    free((char *)g_psDescriptor->Label);
    free((char *)g_psDescriptor->Name);
    free((char *)g_psDescriptor->Maker);
    free((char *)g_psDescriptor->Copyright);
    free((LADSPA_PortDescriptor *)g_psDescriptor->PortDescriptors);
    for (lIndex = 0; lIndex < g_psDescriptor->PortCount; lIndex++)
      free((char *)(g_psDescriptor->PortNames[lIndex]));
    free((char **)g_psDescriptor->PortNames);
    free((LADSPA_PortRangeHint *)g_psDescriptor->PortRangeHints);
    free(g_psDescriptor);
  }
}
```