Muse Design & Implementation

Version 1.0

Muse is hosted on SourceForge:
`http://lmuse.sourceforge.net/`

This document was created using pdfTEX and the ConTEXt macro package.

Contents

1 File Formats

1.1 Instrument Definition Files

Muse Instrument Definition Files are used to describe the properties of external midi instruments.

IDF-Files have the file extension `*.idf` and they are searched in the Muse "instruments" subdirectory (which is on my system `/usr/share/muse-1.0/instruments`.

An instrumentis a property of the midi ouput port . All found instrument definitions are presented by MusE in the Instrument pulldown menu of the midi output port.

Example of an instrument definition file:

```
<?xml version="1.0"?>
<muse version="1.0">
  <MidiInstrument name="GM">
    <Init>
      ...MusE event list which initializes instrument
      </Init>
    <PatchGroup name="Piano">
       <Patch name="Grand Piano" prog="0"/>
       <Patch name="Bright Piano" prog="1"/>
       ...
      </PatchGroup>
    <PatchGroup name="Bass">
      <Patch name="Acoustic Bass" prog="32"/>
      <Patch name="Fingered Bass" prog="33"/>
      </PatchGroup>
    ...
    <Controller name="Brightness">
      <n>0x4a</n>
      <min>0</min>
      <max>127</max>
      <init>0</init>
      </Controller>
    ...
    </MidiInstrument>
  </muse>
```

PatchGroupsare not mandatory; its valid to write:

```
<MidiInstrument name="GM">
  <Patch name="Grand Piano" prog="0"/>
  <Patch name="Bright Piano" prog="1"/>
  <Patch name="Acoustic Bass" prog="32"/>
  <Patch name="Fingered Bass" prog="33"/>
  ...
  </MidiInstrument>
```

An instrument definition file should only define one instrument.

The "prog" parameter in a "Patch" is mandatory. Missing "hbank" or "lbank" are treated as "don't care". Missing "drum" is treated as drum="0".

A XG-Patch looks like this:

```
<Patch name="Electro" drum="1" hbank="127"
   lbank="0" prog="24"/>
```

A patch can be associated with a "mode" with one of:

1 - GM
2 - GS
4 - XG

Example:

```
<Patch name="Electro" mode="4" drum="1" hbank="127"
   lbank="0" prog="24"/>
```

Mode id's can be ore'd together for patches which are valid for more than one mode:

```
<Patch name="Grand Piano" mode=7" hbank="0" lbank="0" prog="0"/>
```

example for MusE event (Sysex "XG-On"):

```
<event tick="0" type="5" datalen="7">
  43 10 4c 00 00 7e 00
  </event>
```

`<Init> ... </Init>` part can be omitted

Controllerhave the following properties:

name: arbitrary unique (short) string describing the controller
n: controller number, defines also the controller type:
values from `0x0 - 0x7f` are 7Bit controller
values from `0x1000 - 0x1ffff` are 14 bit controller with MSB/LSB value

pairs
values from `0x20000 - 0x2ffff` are RPN's
values from `0x30000 - 0x3ffff` are NRPN's

min: minimum value for controller

max: maximum value for controller

init: reset value for controller; when controller value is undefined after instrument reset, use the "undefined" value `0x10000`

the min/max/init values can be ommited

2 Plugins

2.1 Midi Plugins

Midi plugins have the file extension `*.so` and they are searched in the Muse "`midiplugins`" subdirectory (which is on my system `/usr/lib/muse-1.0/midiplugins`.

2.1.1 Midi Filter

Example for an midi plugin which does essentially nothing:

First define your custom `Filter` class with `Mempi` as base class.

```
#include "../libmidiplugin/mempi.h"

class Filter : public Mempi {
      virtual void process(unsigned, unsigned,
         MPEventList*, MPEventList*);

   public:
      Filter(const char* name, const MempiHost*);
      ~Filter() {}
      virtual bool init() { return false; }
      };

Filter::Filter(const char* name, const MempiHost* h)
   : Mempi(name, h)
      {
      }
```

Now comes the main `Mempi` function you have to implement: `process()` which has two parameters, an midi event input list and an midi event output list. The example code simply copies all events from the input list to the output list.

```
void Filter::process(unsigned, unsigned, MPEventList* il,
   MPEventList* ol)
      {
      for (iMPEvent i = il->begin(); i != il->end(); ++i) {
            ol->insert(*i);
            }
      }
```

When loading the plugin, the `Mempi` Host looks for the symbol `mempi_descriptor`. This function returns a pointer to the plugin class description you have to fill in.

```
static Mempi* instantiate(const char* name, const MempiHost* h)
      {
      return new Filter(name, h);
      }

extern "C" {
      static MEMPI descriptor = {
            "Filter",
            "MusE Zero Filter",
            "0.1",                  // filter version string
            MEMPI_FILTER,           // plugin type
            MEMPI_MAJOR_VERSION, MEMPI_MINOR_VERSION,
            instantiate
            };

      const MEMPI* mempi_descriptor() { return &descriptor; }
      }
```

2.1.2 GUI interface

Every midi plugin can implement a native graphical user interface (GUI). For this you have to overload five virtual functions.

```
   virtual bool hasGui() const { return true; }
```

This function informs the host whether there is an GUI or not.

```
   virtual bool guiVisible() const { return gui->isVisible(); }
```

This fuction returns `true` if your GUI is visible.

```
   virtual void showGui(bool val) const { gui->setShown(val); }
```

This function shows or hides the GUI.

There are two helper functions the plugin host uses to save and restore the GUI geometry across sessions:

```
   virtual void getGeometry(int* x, int* y, int* w, int* h) const;
   virtual void setGeometry(int, int, int, int);
```

2.1.3 Persistent Parameters

The plugin host can save and restore plugin parameter across sessions. When the host saves his session status, he calls the midi plugin function:

```
virtual void getInitData(int* len, const unsigned char**p) const;
```

You can overload this function to return a pointer to your session data and the len of the data block.

When the session is restored, the host calls

```
virtual void setInitData(int len, const unsigned char* p);
```

which you can use to restore the last plugin state.

2.1.4 Host Access Functions

The plugin host supplies some functions which are accessible through the `MempiHost*` in the `Mempi` interface.

The host supplies the following functions:

```
int division() const;
```

returns the midi `division` value which gives the number of midi ticks per midi beat, typically 384.

```
int tempo(unsigned tick) const;
```

returns the midi tempo value.

```
unsigned tick2frame(unsigned tick) const;
```

This function converts a transport position given in midi ticks to a audio frame position.

```
unsigned frame2tick(unsigned frame) const;
```

This function converts a transport position given in audio frames to midi ticks.

```
void bar(int tick, int* bar, int* beat, unsigned* rest) const;
```

This function return bar/beat and a tick rest for a given tick position.

```
unsigned bar2tick(int bar, int beat, int tick) const;
```

This is the opposite to the `bar` function. It converts a time position given in bar/beat/tick into a midi tick value.

2.1.5 Midi Synthesizer

As an example for a midi synthesizer plugin we use a midi metronome.

Lets start with the class definition:

```
#include "../libmidiplugin/mempi.h"

//---------------------------------------------------------
//   metronom - simple midi metronom
//---------------------------------------------------------

class Metronom : public Mempi {

   protected:
      struct InitData {
            char measureNote;
            char measureVelo;
            char beatNote;
            char beatVelo;
            } data;
      MetronomGui* gui;
      friend class MetronomGui;

      unsigned int nextTick;
      unsigned int lastTo;

      virtual void process(unsigned, unsigned, MPEventList*,
         MPEventList*);

   public:
      Metronom(const char* name, const MempiHost*);
      ~Metronom();
      virtual bool init();

      virtual bool hasGui() const      { return true;             }
      virtual bool guiVisible() const  { return gui->isVisible(); }
      virtual void showGui(bool val)   { gui->setShown(val);      }
```

```
    virtual void getGeometry(int* x, int* y, int* w, int* h) const;
    virtual void setGeometry(int, int, int, int);

    virtual void getInitData(int*, const unsigned char**) const;
    virtual void setInitData(int, const unsigned char*);
    };
```

3 The M.E.S.S (MusE Experimental Software Synthesizer) API

M.E.S.S.stands for MusE Experimental Soft Synth and is an API/interface (like DSSI) to enable programmers to easily create their own softsynths.

The interface for a M.E.S.S.-synth consists of two classes, the Mess and MessGui. In short, the framework provides the Mess (the synth) midi data (notes, controllers and sysexes) and methods that handle the IPC between the gui and the synth, so the programmer can focus on DSP-issues instead.

The interface for the synth is specified in the mess.h file in the libsynti directory. The interface for the MessGui is specified in the gui.h file.

```
//---------------------------------------------------------
//  Mess
//    MusE experimental software synth
//    Instance virtual interface class
//---------------------------------------------------------

class Mess {
      MessP* d;

      int _sampleRate;
      int _channels;                // 1 - mono,  2 - stereo

   public:
      Mess(int channels);
      virtual ~Mess();

      int channels() const       { return _channels;   }
      int sampleRate() const     { return _sampleRate; }
      void setSampleRate(int r)  { _sampleRate = r;    }

      virtual void process(float** data, int offset, int len) = 0;

      // the synti has to (re-)implement processEvent() or provide
      // some of the next three functions:

      virtual bool processEvent(const MidiEvent&);
      virtual bool setController(int, int, int) { return false; }
```

```
      virtual bool playNote(int, int, int) { return false; }
       virtual bool sysex(int, const unsigned char*) { return false;
}

      virtual void getInitData(int*, const unsigned char**) const {}
      virtual int getControllerInfo(int, const char**, int*, int*, int*)
const {return 0;}
      virtual const char* getPatchName(int, int, int) const { return
"?"; }
      virtual const MidiPatch* getPatchInfo(int, const MidiPatch*) const
{ return 0; }

      // synthesizer -> host communication
      void sendEvent(MidiEvent);  // called from synti
      MidiEvent receiveEvent();   // called from host
      int eventsPending() const;

      // GUI interface routines
      virtual bool hasGui() const { return false; }
      virtual bool guiVisible() const { return false; }
      virtual void showGui(bool) {}
      virtual void getGeometry(int* x, int* y, int* w, int* h) const;
      virtual void setGeometry(int, int, int, int) {}
      };
```

3.1 Overview of virtual member functions

This is a short description of the virtual functions in the Mess-class:

3.1.1 process

```
virtual void process(float** data, int offset, int len) = 0;
```

This is where the audio processing takes place. The Mess is called with a float array, and it's up to the programmer to fill it with data beginning at data[offset] to data[offset+len].

3.1.2 processEvent

There are two ways to go when it comes to how midi data is fed into the Mess. You can either implement processEvent and manage the different event types (controllers,

sysexes and notes) or implement one or more of the setController, playNote and sysex functions.

Note that overriding processEvent results in setController, playNote and sysex not being called!

3.1.3 setController, playNote, sysex

The setController, playNote and sysex functions should return false if the event was processed, true if the synth was busy.

```
virtual bool setController(int channel, int controller_no, int value)

virtual bool playNote(int channel, int pitch, int velo)

virtual bool sysex(int length, const unsigned char* data)
```

3.1.4 getInitData

MusE provides an interface for storing the state of the synth in the project file. getInitData is called when the project is saved, and provides a pointer to a data block where you store the data which will recreate the synths current state when the project is reopened.

```
virtual void getInitData(int* nrofchars, const unsigned char** data)
const {}
```

3.1.5 getControllerInfo

getControllerInfo is called when the Mess is instantiated, and this is where you tell MusE which controllers the Mess supports. The function is called repeteadly: just fill in the values for the controller and return the index you want to be sent next time. Return 0 when you've given MusE information about all controllers your Mess supports.

As you can see, the necessary parameters are an index (used internally), a name for the controller, an int describing the actual controller number (f.ex. 91 for reverb), and finally min and max values for the controller.

```
virtual int getControllerInfo(int index,
                              const char** name,
                              int* controller,
```

```
                                int* min,
                                int* max) const
```

3.1.6 hasGui

hasGui is called when the Mess is instantiated, and tells MusE if the synth has a gui or not.

```
virtual bool hasGui() const;
```

3.1.7 showGui

showGui is called from MusE when the gui is supposed to be visible

```
virtual void showGui(bool show);
```

3.1.8 getPatchName, getPatchInfo

getPatchName and getPatchInfo are used to give MusE information about patches, which can be selected in the track inspector.

The getPatchName function specifies three parameters to be passed: channel, program, and a third that is unused. It is the responsibility for the synth to return the patchname for the particular channel and a program.

```
virtual const char* SimpleSynth::getPatchName(int channel, int program,
int unused) const
```

getPatchInfo is called repeatedly until the synth returns 0. The synth must send information back in a MidiPatch struct. The MidiPatch parameter sent contains information on the last MidiPatch that was sent from the synth (a null pointer on first call).

```
virtual const MidiPatch* getPatchInfo(int channel,
                                      const MidiPatch* previouspatch)
                                      const;

struct MidiPatch {
      signed char type;   // 1 - GM  2 - GS  4 - XG
      signed char hbank, lbank, prog;
      const char* name;
      };
```

3.1.9 Descriptor

Since the synths are built as shared libraries, you need to specify a descriptor for your MESS. They look something like this: (example taken from SimpleSynth/SimpleDrums)

```
static Mess* instantiate(int sr, QWidget*, const char* name)
 {
  printf("SimpleSynth sampleRate %d\n", sr);
  SimpleSynth* synth = new SimpleSynth(sr);
  if (!synth->init(name)) {
    delete synth;
    synth = 0;
    }
  return synth;
 }
extern "C"
 {
  static MESS descriptor = {
    "SimpleSynth",
    "Mathias Lundgren (lunar_shuttle@users.sf.net)",
    "0.1", //Version string
    MESS_MAJOR_VERSION, MESS_MINOR_VERSION,
    instantiate,
  };
  const MESS* mess_descriptor() { return &descriptor; }
 }
```

3.2 MessGui

For the Gui, there's really only one method that is interesting, processEvent. processEvent in the gui is called whenever the synth sends it a midiEvent, which it does with send(MidiEvent). Since this is the only method of communicating the gui can send midimessages to the synth via sendEvent, sendController and sendSysex calls.

```
virtual void processEvent(const MidiPlayEvent&) {};
```

4 Index