FluidSynth

Performance measurement (Profiling)
Adding "profiling" interface functionality

Ceresa Jean-Jacques

FluidProfile_0001 First writing 15/02/2016. For version 1.1.6
- This patch integrates FluidVoiceOff- 0001

FluidProfile_0002 First writing 04/03/2016. For version 1.1.6
- This patch integrates FluidVoiceOff- 0001
- Minor correction in patch and hardware addition in pdf (see 3.5).

FluidProfile_0003 11/06/2017: replace FluidProfile_0002. For version 1.1.6
- This patch integrates FluidVoiceOff- 0002
- Minor correction in patch and hardware addition in pdf (see 3.5).

FluidProfile_0004 11/02/2018: For version 2.0
- cpu load precision of 1/1000 % for fast CPU.
- adding profiling cancellation key <cr>.
- compensate gain during notes generation.

-

1. Introduction

This document describes a console interface addition for FluidSynth performance measurement (profiling).

Chapter 2 describes actual support available in FluidSynth (v 1.1.6) for library profiling. This chapter is mainly useful for developers. The interesting informations are absolute **duration** and **cpu load.**

- Part of code uner ***duration*** measurement allows developer to compare different algorithm duration whatever hardware are used.
- ***cpu load*** is an other way to reveal duration relative to ***audio period*** on audio output. This is a way to give awswers to the following questions:
  - "What is the proportion of time consumed by the CPU for rendering 10 musical notes?".
  - "How many ***voices*** can be played with this CPUx or with an other CPUy ?"
  - For a same **library** version, ***cpu load*** is a way to compare performance of different **hardware**.
  - On the same **hardware**, ***cpu_load*** is a way to compare different **algorithm** duration when implementing functions.

This chapter is useful for developers who intend to use theses methods, for example to add measurement points (i.e probes) (2.2.8) when necessary.

Version v 1.1.6, is without interactive interface.
Chapter 3 describes a console interface to improve support. With this addition, any console user (end user or developer) has new profile commands allowing easy performance measurement.

- Chapter 3.1 is the user manual for these new commands for any user (developer or end user).
- Chapter 3.2 gives details on the patch content and behaviour. It is intended for developers.
- Chapter 3.3 describes how to apply the patch **0001-profiling-0004-for-v2.0.patch**.

Conclusion:
With the help of these new console commands, any user can contribute to publish a list of hardware performance measurement. This can be useful for "embedded" applications. Chapter 3.5 is a starting place to publish this list.

2. Performance measurement inside FluidSynth

This chapter describes actual support available in FluidSynth (v 1.1.6) for library profiling.This chapter is mainly useful to developers.

The support allows duration measurement of part of code. With this support one can do time measurement of audio rendering functions (see 2.2).
This support allows also time measurement of the input MIDI code (MIDI API) (see 2.1, and 2.2.5).

- "MIDI input" code can be measured with "verbose" mode (see 2.1).
- " Audio rendering API" can be measured with "cpu load" measurement (2.2.1) and "Profiling" added probes code (see 2.2.3).

2.1. Measurement with "verbose" option

"Verbose mode" is useful for time measurement of MIDI API code:
This mode is enabled with the setting "**synth.verbose**". The "code probe" is already in the library. There is no need to configure with profiling option.

Measurement is done with the **fluid_curtime()** function who has 1ms precision.

This mode displays on the console the date of occuring MIDI messages noteOn/Off. Also the date of allocated voices are displayed. It is possible to deduce duration of voice allocation code which is the difference between 2 consecutive displaying.

- **new_fluid_synth()**, is used to initialize a reference date (***start*** in ms), at synthesizer creation.
- **fluid_synth_noteon_LOCAL()**, is used to catch "***noteOn date***" relative to *start* time.
- **fluid_synth_noteoff_LOCAL()**, is used to catch "*noteOff date*"relative to *start* time.
- **fluid_synth_alloc()**,is used to catch "***voice allocation date***" relative to *start* time.

2.2. "audio rendering" API performance measurement

2.2.1. "CPU load" measurement.

This measurement is done with **fluid_utime()** function who has 1 µs resolution.
This measurement is done all the time inside the following audio rendering functions API:
fluid_synth_nwrite_float(), fluid_synth_write_float(), fluid_synth_write_s16().

Further, the value can be read with the function **fluid_synth_get_cpu_load()** API.
This API allows hardware performance measurement in real time mainly useful for vue meter displaying.

2.2.2. "CPU load" definition: **duration relative to sample period in percent**

cpu load is defined as the ratio between the processing time of one sample and the period of this sample outside the audio card. The result is normalized in percent.

cpu_load (%) = (processing time of one sample / period of one sample) x 100

2.2.3. Measurement with WITH_PROFILING macro

This method behaves the same than "**verbose**" option (2.1). It allows to insert a "**macro probe**" inside the part of code under measurement (see 2.2.4) . However, in "**verbose**" mode (see 2.1) , "verbose insertion" is done at execution time (i.e enabled by the setting "**synth.verbose** "). When using "macro probe" , insertion is done at Cmake time choosing **enable-profiling** option (this will define the macro **WITH_PROFILING**). Thus, is is always possible to build a library with full performance (i.e without the profiling added code).
Note that the presence of "macro probe" introduces a very low overload, however for embedded hardware it is usually preferable to re-build without WITH_PROFILING to get rid of unnecessary code.

This measurement is done with the function **fluid_utime()** who has 1 µs resolution.

Warning: Chapiter 3.4 gives important details about the expected precision of this function.

2.2.4. Measurement point "**macro probe profiling**"

The following are macros (enabled by WITH_PROFILING set to 1)

- **fluid_profile_ref(), fluid_profile_ref_var()** allows to get a reference time (in µs).
  This macro needs to be inserted at the beginning part of code to be measured.
- **fluid_profile(_num,_ref)**.
  This macro needs to be inserted at the end part to be measured. It makes the difference time between the end and the begin (delta). The *defta* time is accumulated int the data table ***fluid_profile_data[ ]*** at **_num** entry which is an identifier of the code under measurement..

So both macros **fluid_profile_ref_var(_ref), fluid_profile(_num,_ref)** (in fluid_sys.h) , allows measureament and registration in fluid_profile_data[ ] table (in fluid_sys.c). This table will be used later for displaying (2.2.6).
Each entry in this table is a structure identifying the part of code under measurement.

```
typedef struct _fluid_profile_data_t {
  int num;                    // Part code identifier (voir 2.2.5)
```

```
  char* description;          // name describing the part of code
  double min, max, total;     // duration min, max et total
  unsigned int count;         // number of times the macro has been called
} fluid_profile_data_t;
```

The table ***fluid_profile_data[ ]*** is initialized in fluid_sys.c.

2.2.5. "profiling" Code identifier.

Following are actual "Part of code" identifiers (v 1.1.6) (fluid_sys.h).

Following identifiers are for "Audio rendering" API:

- *Duration of **fluid_synth_write_float()** or **fluid_synth_write_s16()** or fluid_synth_dither_s16()*
  FLUID_PROF_WRITE
- *Duration of **fluid_synth_render_blocks()**.*
  FLUID_PROF_ONE_BLOCK
- *Duration of **clearing buffers** in **fluid_rvoice_mixer_render()***
  FLUID_PROF_ONE_BLOCK_CLEAR
- *Duration of **fluid_mixer_buffers_render_one()** (for one voice)*
  FLUID_PROF_ONE_BLOCK_VOICE
- *Duration of **fluid_render_loop_singlethread()** or **fluid_render_loop_multithread()***
  FLUID_PROF_ONE_BLOCK_VOICES
  time of **fluid_rvoice_mixer_render(),** without *fluid_rvoice_mixer_process_fx() ([reverb] + [chorus])*
- *Duration of **fluid_rvoice_mixer_process_fx()** (reverb only).*
  FLUID_PROF_ONE_BLOCK_REVERB,
- *Duration of **fluid_rvoice_mixer_process_fx()** (chorus only)).*
  FLUID_PROF_ONE_BLOCK_CHORUS,

Following identifiers are for "MIDI" API

- FLUID_PROF_VOICE_NOTE *time between fluid_voice_start() and fluid_voice_noteoff()(see R1)*
- FLUID_PROF_VOICE_RELEASE *time between fluid_voice_start() and fluid_voice_off() (R2,R3)*

R1: Note duration until note Off.
R2: Note duration until end of release.
R3: Release duration is:

Release = FLUID_PROF_VOICE_RELEASE - FLUID_PROF_VOICE_NOTE

2.2.6. Profiling displaying

Informations measurement are recorded in fluid_profile_data[] during the synthesizer life. Results are displaying with **fluid_profiling_print()** at destruction time(delete_fluid_synth()). The function code exists only if WITH_PROFILING MACRO is defined. The function is defined in fluid_sys.c. Text format follows:

```
fluid_profiling_print
fluidsynth: Estimated times: min/avg/max (micro seconds)
fluidsynth: fluid_synth_write_*              : min / average  / max
fluidsynth: fluid_synth_one_block            : min / average  / max
fluidsynth: fluid_synth_one_block:clear      : min / average  / max
fluidsynth: fluid_synth_one_block:one voice: min / average  / max
fluidsynth: fluid_synth_one_block:all voices : min / average  / max
fluidsynth: fluid_synth_one_block:reverb     : min / average  / max
fluidsynth: fluid_synth_one_block:chorus     : min / average  / max
fluidsynth: fluid_voice:note                 : min / average  / max
```

fluidsynth: fluid_voice:release : min / average / max

2.2.7. Notes about "profiling" measurement points

This chapter gives details about measurement points and internal functions concerned.
Remarks:

- *Duration* of ***fluid_synth_write_s16(), fluid_synth_write_float()***
  FLUID_PROF_WRITE = FLUID_PROF_ONE_BLOCK + writting in buffers of the caller

  Writting in buffers of the caller = FLUID_PROF_WRITE - FLUID_PROF_ONE_BLOCK
- *Duration* of **fluid_synth_render_blocks()**. (number of blocks FLUID_BUFSIZE)
  FLUID_PROF_ONE_BLOCK = dispatch_all() + timer_process() +
  fluid_rvoice_mixer_render() (FLUID_PROF_ONE_BLOCK_VOICES + [Reverb] + [Chorus])

- *Duration* of **fluid_rvoice_mixer_render(),** (All voices on a number of blocks FLUID_BUFSIZE)
  Durée **fluid_rvoice_mixer_render() = FLUID_PROF_ONE_BLOCK_VOICES** +
  [FLUID_PROF_ONE_BLOCK_REVERB]
  [FLUID_PROF_ONE_BLOCK_CHORUS]

  FLUID_PROF_ONE_BLOCK_VOICES , mono thread or multithread (without reverb et chorus)
  Useful f to compare:
  - support mono / multi thread.
  - compute voice duration (based on voices number knowledge) and compare with FLUID_PROF_ONE_BLOCK_VOICE.

  Remark: see note in **FLUID_PROF_ONE_BLOCK_VOICE** about dependency of fx unit.

- *Duration* of ***fluid_mixer_buffers_render_one()*** *(*One voice on a number of blocks FLUID_BUFSIZE*)*.
  FLUID_PROF_ONE_BLOCK_VOICE

  Note: Normally this duration should be independent of effect unit presence(reverb,chorus).
  however, the send parameter (for reverb or chorus) is computed only if the corresponding buffers are prepared in fluid_**mixer_buffers_prepare()** and used **fluid_rvoice_buffers_mix()**, so the duration **FLUID_PROF_ONE_BLOCK_VOICE** and **FLUID_PROF_ONE_BLOCK_VOICES** are a bit dependent of presence of reverb or chorus fx unit.

- Time of **fluid_rvoice_mixer_process_fx()** (reverb. only) (on a number of blocks FLUID_BUFSIZE)
  FLUID_PROF_ONE_BLOCK_REVERB

- Time of **fluid_rvoice_mixer_process_fx()** (chorus only) (on a number of blocks FLUID_BUFSIZE)
  FLUID_PROF_ONE_BLOCK_CHORUS

2.2.8. Adding a new "profiling" point

If one wants to add a new measurement point, the steps ares::

- Add an *entry* in ***fluid_profile_data[ ]*** table *(fluid_sys.c)* and a new value in ***enumeration*** (see 2.2.5) (each value is an entry index in the table).
- Add points using **fluid_profile_ref()** or **fluid_profile_ref_var()** macro at the *beginning part* and **fluid_profile(_num,_ref)** macro at the *end part* (2.2.4).

3. Adding - profiling commands interface

This chapter describes a console interface to improve profiling support. With this addition any console user (end user or developer) has a new set of commands allowing easy performance measurement.

- Chapter 3.1 is the user manual for these new commands (useful for any user).
- Chapiter 3.2 gives details on the patch contents and behavior (useful for developer).

3.1. New "performance profiling" commands set

This command set adds functionality to the actual support described in 2.2.

A new set of "profile" commands is very useful to do hardware performance measurement. This allows **cpu load** evaluation (**total(%)**) for a given number of voices (**nVoices**). So one can estimate the maximum number of voices (**maxVoices**) this hardware could generate.

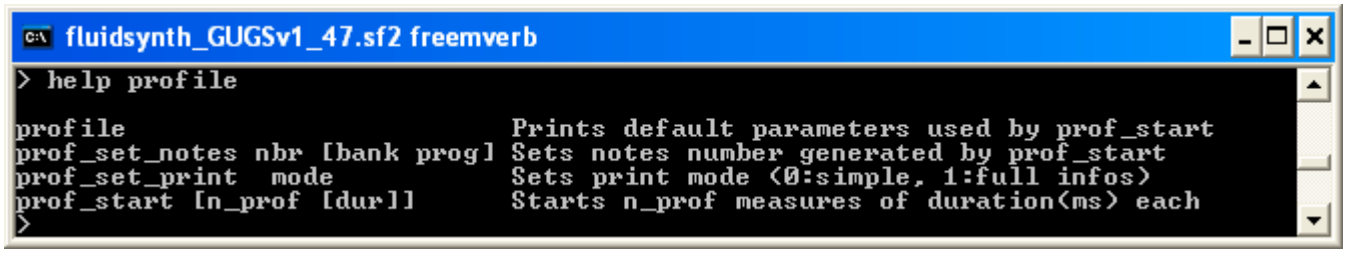

Fig.1

With the help of interactive interface, the user chose:

- **profile** command allows to print default parameters used by the **prof_start** command. (3.1.1).
- **prof_set_print** command allow to choose *printing mode* (see 3.1.2).
- The window measurement (**n_prof** and **duration**) (see **prof_start** command see 3.1.6).

Results displaying is done on the console screen (see 3.1.2).

- Input sources MIDI events could be:
  - A MIDI file (see 3.1.3) or
  - A constant number of notes (**prof_set_notes** command, (see 3.1.4).

3.1.1. Displaying default parameters: **profile**

The default parameters are those used by the **prof_start** command (see 3.1.6).
The **profile** command display default parameters:

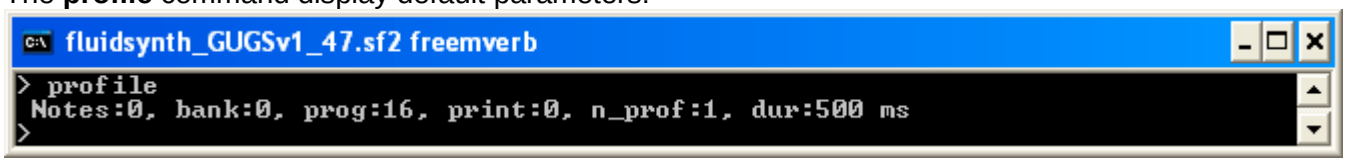

Fig.2

- **Notes**, **bank** and **prog** can be changed by the **prof_set_notes** command (see 3.1.7).
- **print mode** can be changed by the **prof_set_print** command (see 3.1.2)
- **n_prof**, **dur** can be changed by the **prof_start** command (see 3.1.6)

3.1.2. Printing results on console – **print mode**

Here is an example displayed by **prof_start** command**.**

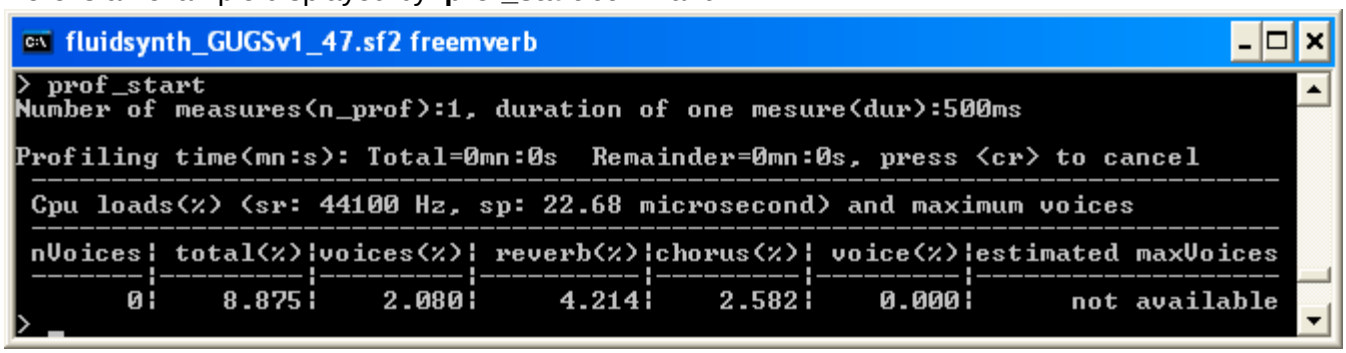

Fig.3: Example with no MIDI messages received. No voices are played.

In this example (Fig.3), the measurement window is 1 measure (default) with 500 ms width. Total duration is 0,5s.
On each result, *total* duration and *remainder* duration are displayed in **minutes:secondes**

Printing is mode 0 (default) who displays only "cpu load".
This mode is often enough to estimate hardware performance.

Each result have followings values:

- **nVoices**: average voices number actually playing.
- **total(%)**: average total cpu load (voices% + reverb% + chorus%) in percent.
- **reverb(%)**: average reverb cpu load in percent.
- **chorus(%)**: average chorus cpu load in percent.

Following values are computed from measurement for estimations.

- **voices(%)**: average all voices cpu load in percent (without Reverb, without Chorus) : voices% = total% - reverb% - chorus%.
- **voice(%)**: average one voice cpu load in percent . The value is computed as this: voice = **FLUID_PROF_ONE_BLOCK_VOICES / nVoices.**
- **estimated maxVoices**: Estimation of maximum number of voices this hardware could generate (i.e assuming 100% CPU , without reverb and without chorus). This value is computed as this: maxVoices= (100% - reverb% - chorus%) / voice%.

To obtain a full display, the user need to change the print mode using **prof_set_print** command.

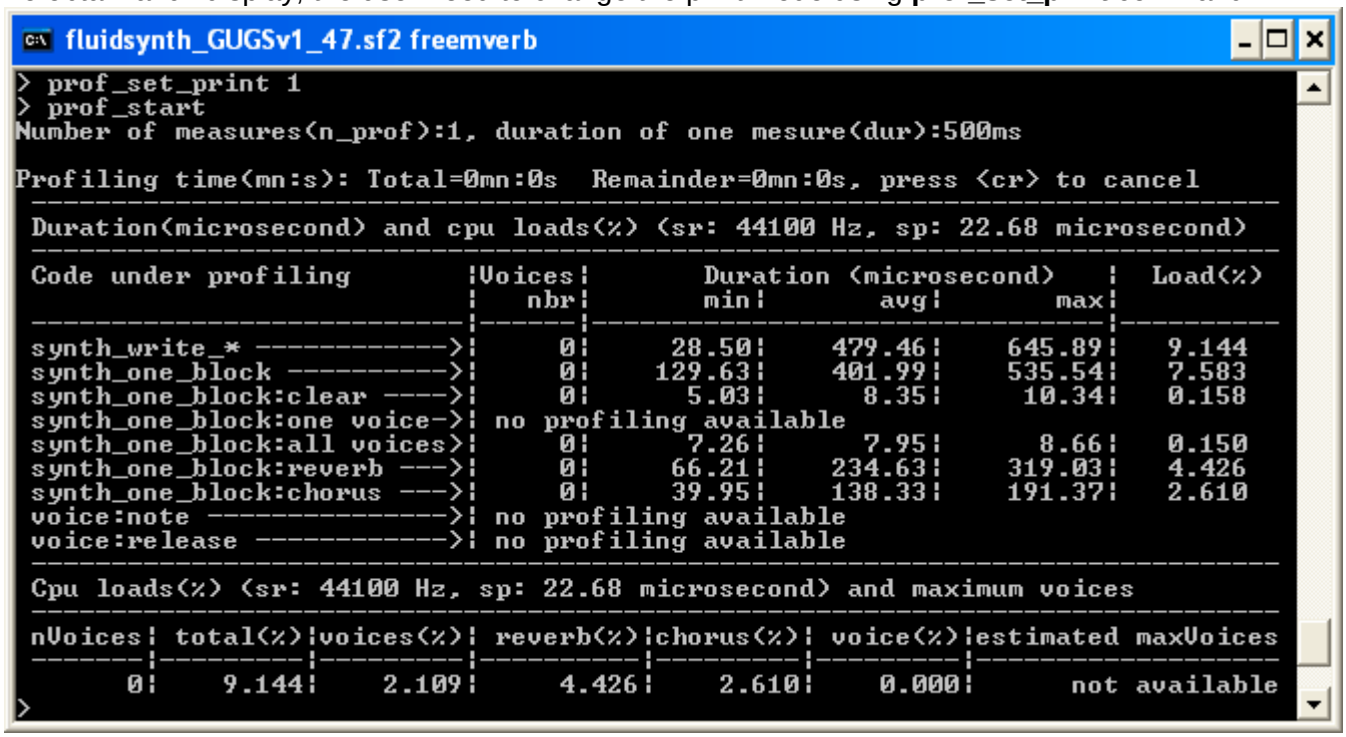

Fig. 4: Example with no MIDI messages received. No voices are played.

In this example (Fig.4), printing mode is set to 1. This mode is mainly useful for developers for code measurement / optimisation efficiency.
In mode 1, informations displayed are those of mode 0 (see Fig.3), with a preceding table of durations (in µs) and cpu load (in %) for all measurements code described in chapter 2.2.5. Each column describes following values:

- **code** identify the code under measurement (see 2.2.5).
- **Voices nbr**: average voices number.
- **Duration (µs)**: duration, min/avg/maximum.
- **Load(%)**: cpu load in percent (see the definition in 2.2.2).

3.1.3. Profiling when playing MIDI file

When a MIDI file is playing, the shell allow to start a burst measurement at any time while listening using **prof_start** command (3.1.6).
This kind of measurement allows estimation of total cpu load (**total(%)**) and actives voices number (**nVoices**). However, as the numbers of notes varies from one measure to the other, this kind of measurement is not precise. To get precise measurement see 3.1.4.

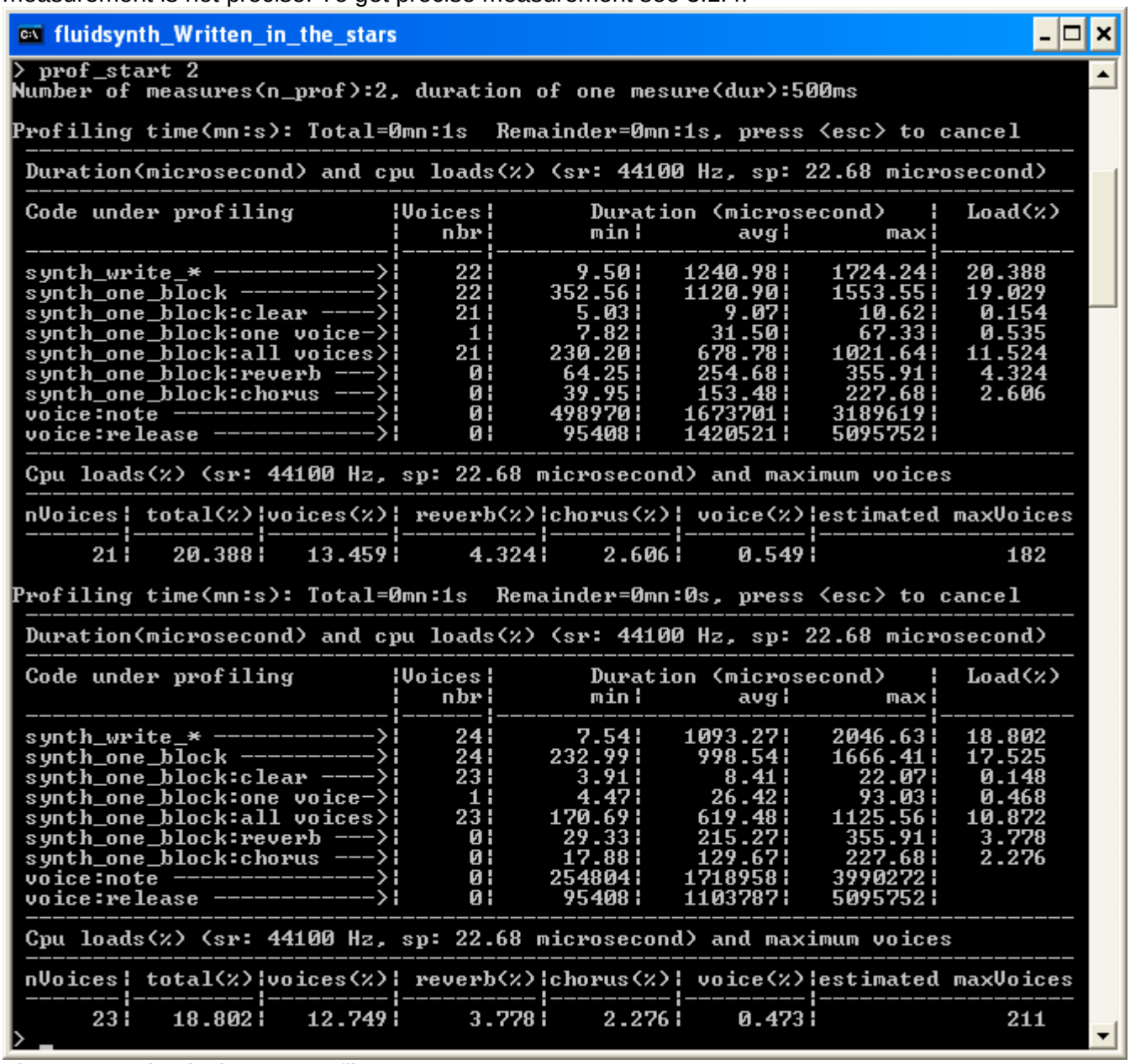

Fig.5: Example playing a MIDI file

3.1.4. Precise performance measurement

To get a precise cpu load per voice (**voice(%)**) and to get a maximum number of voices (**estimated maxVoices**), the shell allows to choose constant number of notes that will be generated during profiling (see **prof_set_notes** 3.1.7).
In this case, playing a MIDI file is not necessary and unuseful. Notes will be generated automatically by the **prof_start** command (3.1.6).
As the user can choose constant number of notes, the number of voices generated will be constant (see 3.1.5).

3.1.5. Useful preset for precise profiling: GUGSv1_47.sf2 – bank:0 prog:16

To be sure that voices number remains constant, voices must not vanish during profiling. To get this result the soundfont preset used needs to be well suited.

The best preset needs to have the following design:
Volume enveloppe ADSR must be:

- Delay: 0
- Attack: very short
- Hold: 0
- Decay: no decay
- Sustain 100 %
- Release: very short.

No decay: this choose is important because when the voice amplitude reachs 0, the voice is automatically free by the synthesizer.The **prof_set_notes** command allow to choose bank and prog preset number (see 3.1.7).

This preset is a good candidate: **GUGSv1_47.sf2, preset organ1 (bank:0 prog:16)**

3.1.6. Starting /Canceling measurement command: **prof_start**

The user starts a burst of measure using this command: **prof_start [n_prof [dur]].**
n_prof, dur parameters are optionals. When there are given they change the default values.

- ***n_prof*** (default 1) and ***dur*** in ms (default 500 ms) are the number of measures and the width duration of one mesure .
- Results are displayed for each measure depending of printing mode (see 3.1.2).

Note: When a measurement has been started with a large value for n_prof or dur, the measurement can be cancelled using <cr> key.

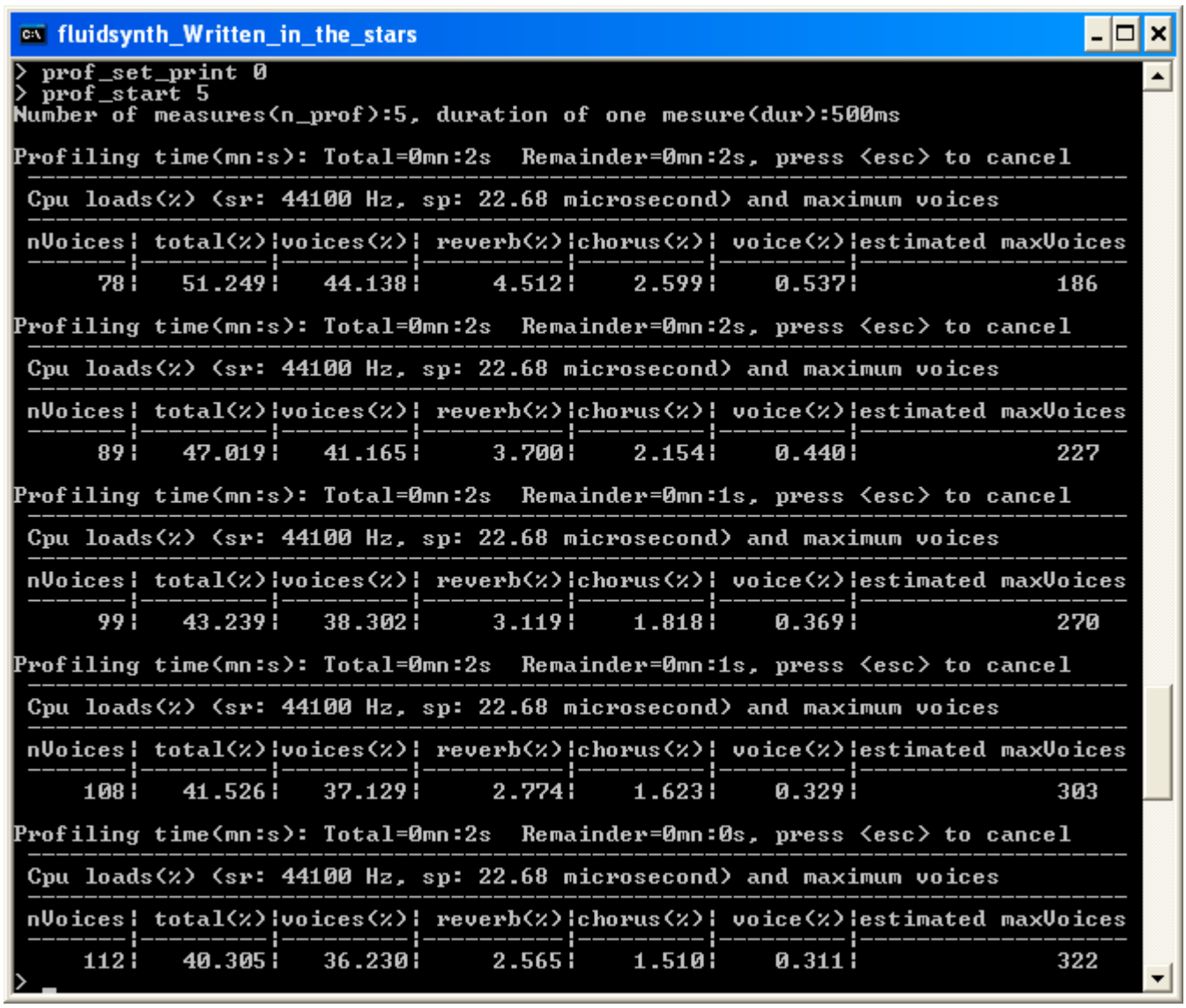

Fig.5: Example playing a MIDI file

This example (Fig.5), a burst of 5 measures (500ms each). Total time is 2,5 s.
When input is a MIDI file, value change for each measure.
The parameters are memorized and become default values for the next command.

Example 1:
>prof_start
Is equivalent to: profile_start 5.

Exemple 2:
>prof_start 10 500
Displays 10 measures of 500ms each.Total time is 5 seconds
The parameters are memorized and become default values for the next command

Exemple 3:
>prof_start
Is equivalent to: profile_start 10 500.

3.1.7. Number of notes to generate: **prof_set_notes**

The **prof_set_notes nbr [bank,prog]** command allows to choose the number of notes that will be generated by the **prof_start** command before starting a burst of measures (3.1.6).
bank prog parameters are optionals. When there are given they change the default values.

- ***nbr*** is the number of notes (0 by default). When 0, no notes will be generated.
- ***bank*** et ***num*** are bank (0 to 127) and preset number (0 to 127) in the soundfont.

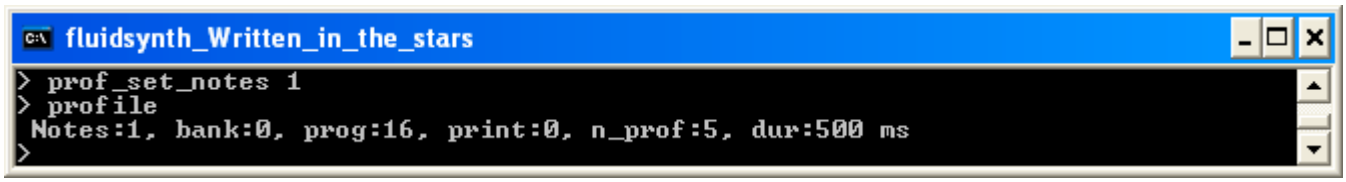

Fig.6: Only one note will be generated by **prof_start** using le preset bank 0 , program 16.

When generating a number of notes, the synthesizer must not already playing voices. Otherwise, generation will be refused and a message is displayed: "**Warning: can't generate notes, stop any playing"** (see Fig.7).

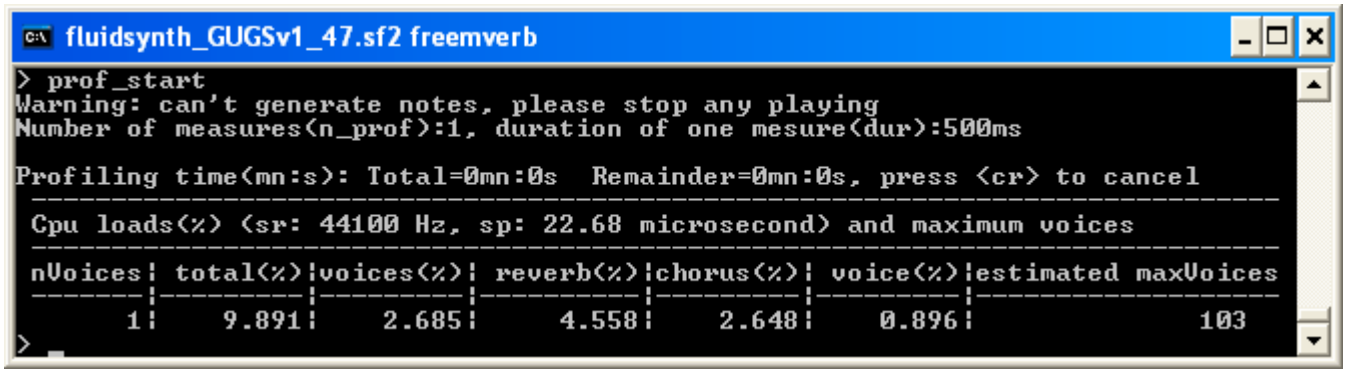

Fig.7: Notes generation is refused because the synthesizer is already playing.

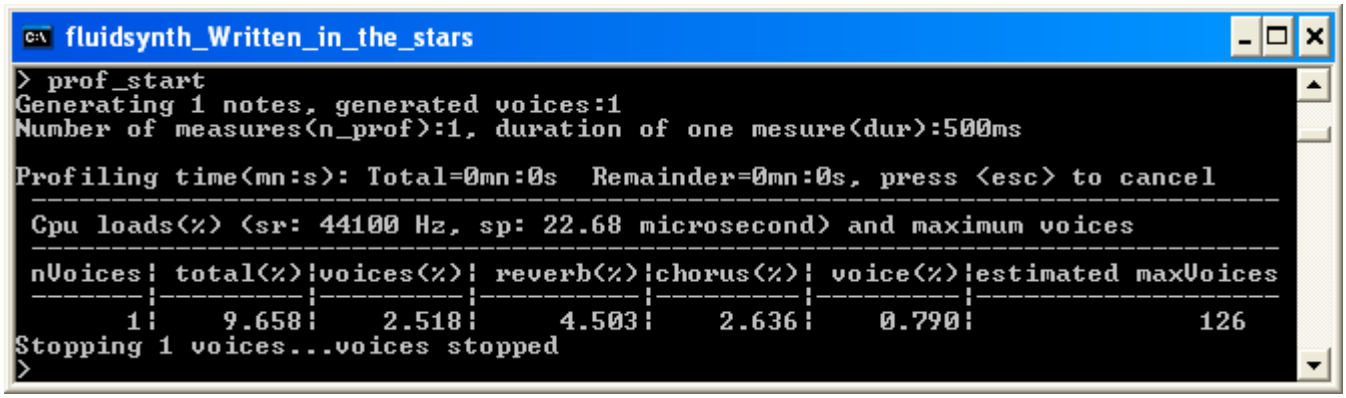

Fig.8:The synthesizer accepts notes generation.

In example Fig 8. When notes are generated, the display is:
"**generating xx notes, generated voices:yy"**

- xx is the number of generated notes choosen by prof_set_notes (3.1.7).
- yy is the number of generated voices that may be different than *xx* depending of the preset composition (key range, and instrument zone layering).

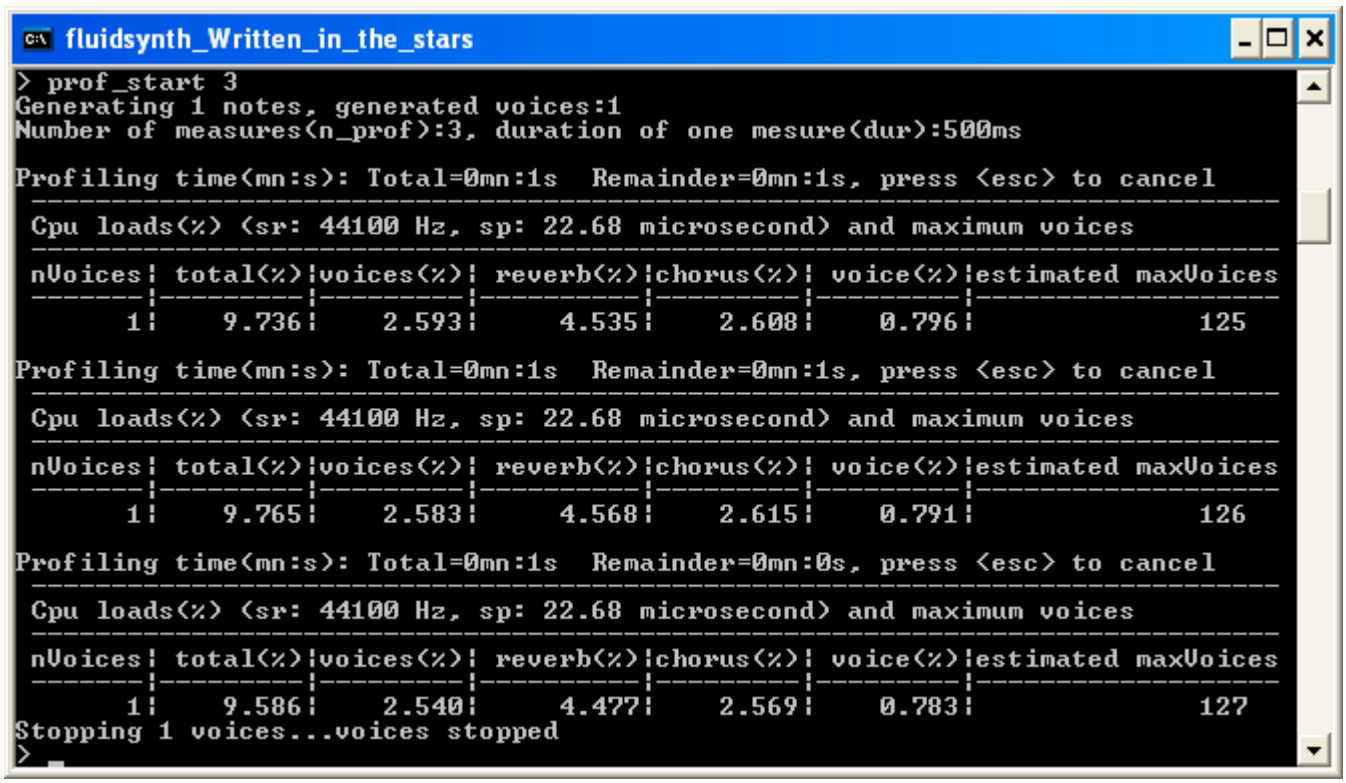

Fig.9

In example Fig.9 The sequence is the following:

- generation of xx notes (i.e 1)
- start of measure 1 , waits and displays result.
- start of measure 2 , waits and displays result.
- ....
- ....
- stops voices generation of yy voices (i.e 1).

Remark: To get a good value for **estimated maxVoices**, it is better to choose 10 notes or above

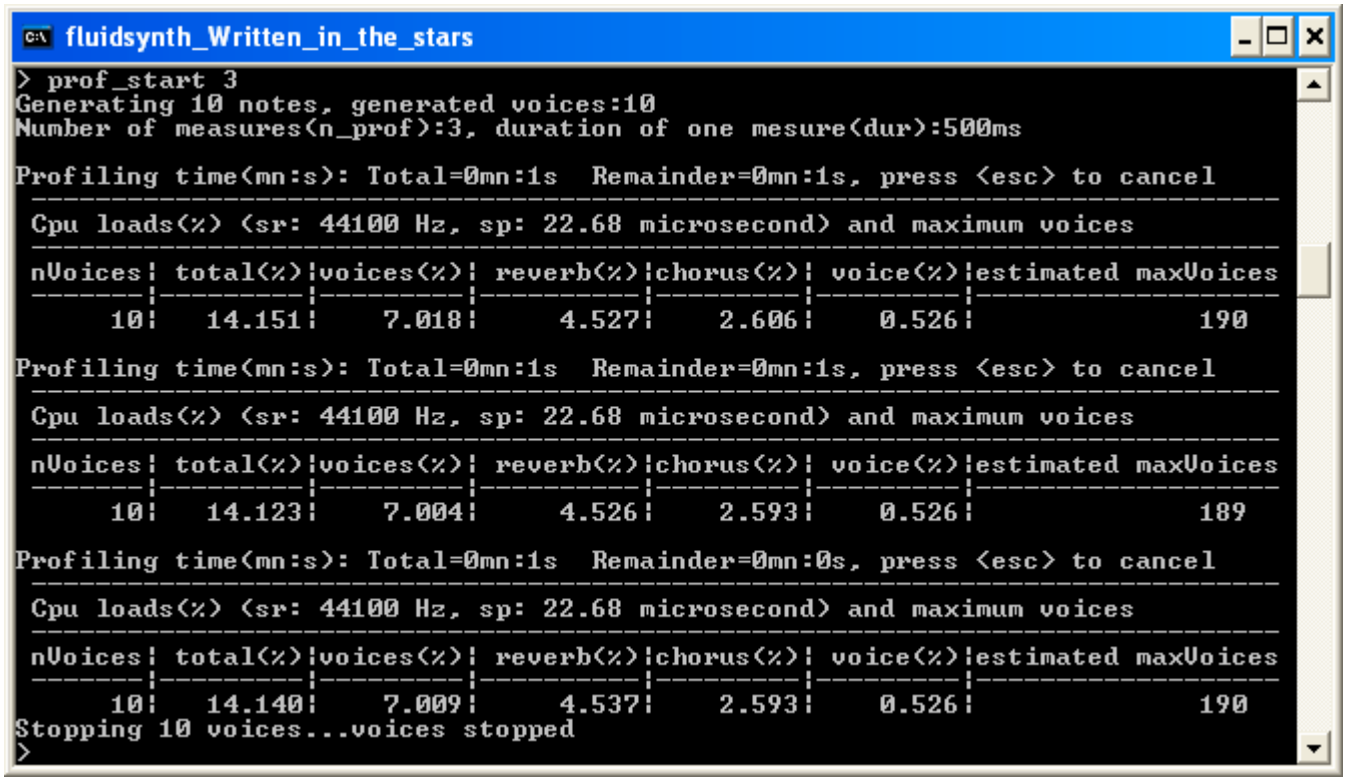

Fig.10: In this example, with 10 notes, total cpu load is 14.14 %. The plateform could play 190 voices (maximum) assuming total load of 100%.

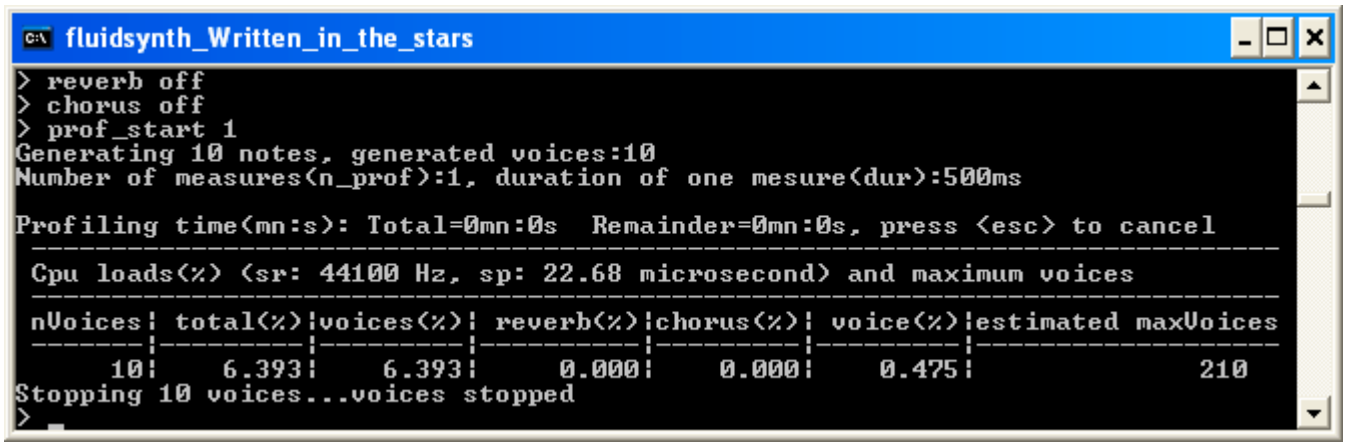

Fig. 11: In this example, without reverb and without chorus, with 10 notes, total cpu load is 6.393 %. The plateform could play 210 voices (maximum) assuming total load of 100%.

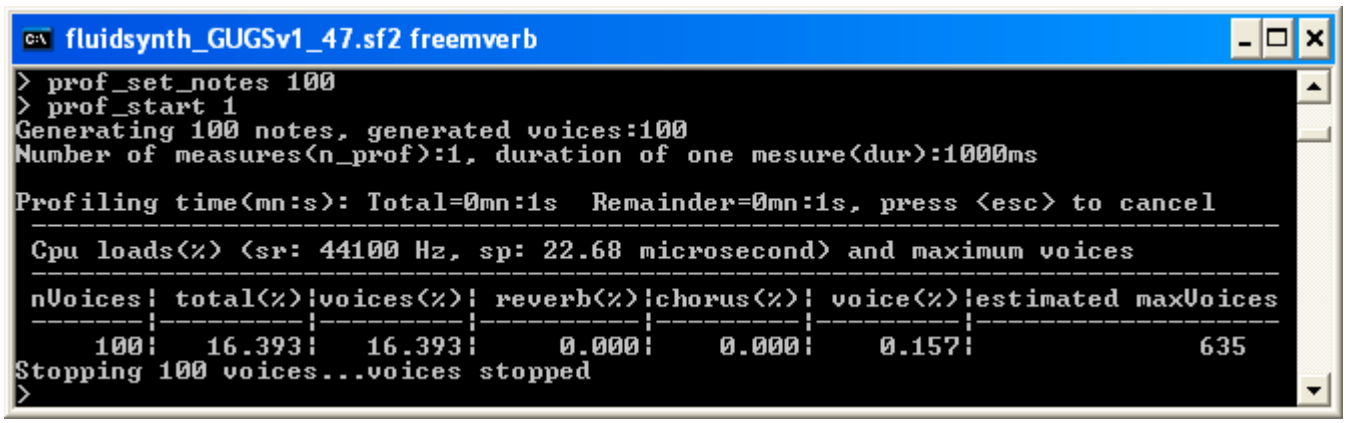

Fig.12: In this example, on an other hardware plateform, without reverb and without chorus, with 100 notes, total cpu load is 16.393 %. The plateform could play 635 voices (maximum) assuming total load of 100%.

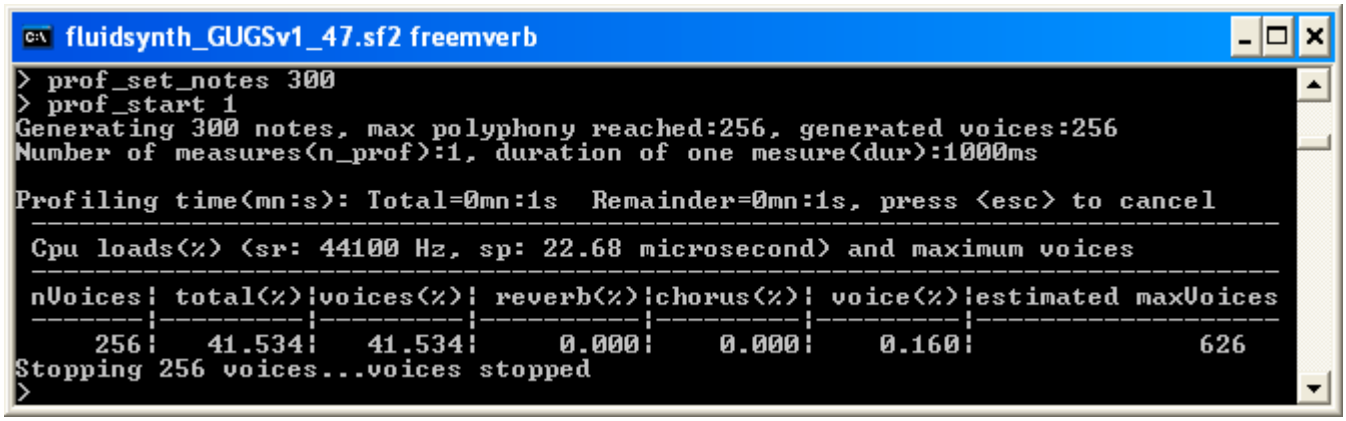

Fig.13: In this example, without reverb and without chorus, with 300 notes, total cpu load is 41.534 %. The plateform could play 626 voices (maximum) assuming total load of 100%.

Notes generation is limited by the setting ***synth.polyphony*** (see Fig.13 , the message is:"**generating xx notes, max polyphony reached:256, generated voices:256"**)

Remark: In all cases, **estimated maxVoices** is the voices number that the plateform could play assuming total load without reverb and without chorus (100% - [reverb% + chorus%]).

3.2. Implementation: adding profiling interactive interface

This chaper is the implementation of the specifications described in chapter 3.1.

3.2.1. overview behaviour

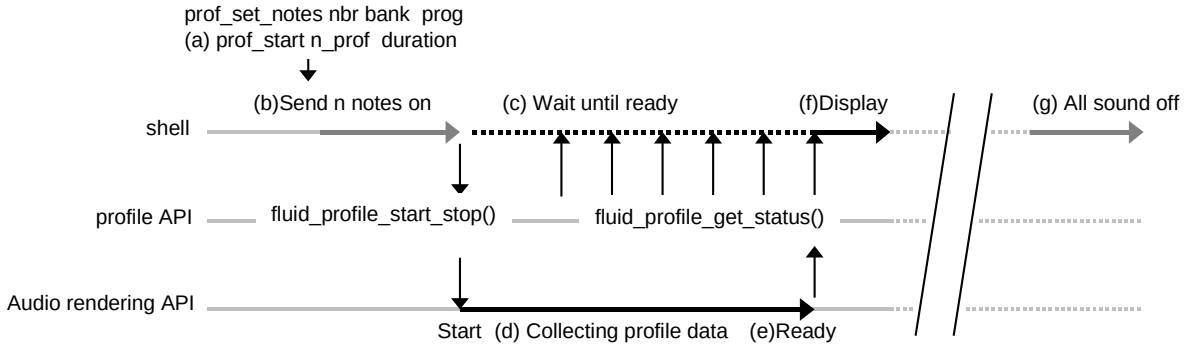

Fig.1

Figure Fig.1 shows how it works
1) The command requests a measurement (a) **prof_start** (3.2.8) in the shell task context and waits the result (c).

2) Then the data are collected (d) in one of theses audio rendering API function:
fluid_synth_nwrite_float() or **fluid_synth_write_float()** or **fluid_synth_write_s16()** each time the function is called (in the audio context task) (see 3.2.15). When measure duration is elapsed, the audio rendering API signals that the data are ready (e).

3) When collected data are ready, shell command (prof_start) prints results (f) (see 3.2.13).

Eventually, notes are generated before the first measure (b) and stopped after *n_prof* measures (g) (see 3.2.9).

We remark, that the audio rendering API doesn't print result but only collect the data. The collect overload is low (see 3.2.15).

So an interface is necessary between **prof_start** command and "audio rendering API" (see 3.2.2).

The existence of this new shell command and new "profiling interface" need to be chosen at compilation time with WITH_PROFILE macro.

3.2.2. Interface between **profile** commands and **audio rendering(fluid_sys.c,h)**

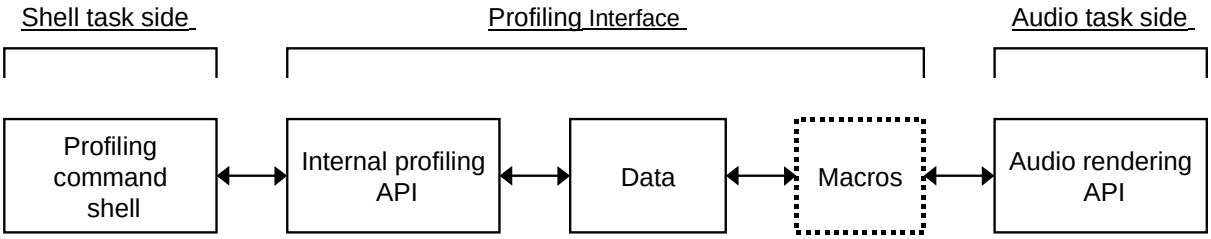

Fig. 3.1

Figure 3.1 shows the "Profiling" interface between shell commands and Audio rendering API.

Th internal profiling API is made of functions **fluid_profile_start_stop()** , **fluid_profile_get_status()** and **fluid_profile_print_data()**.

Fig.3.2

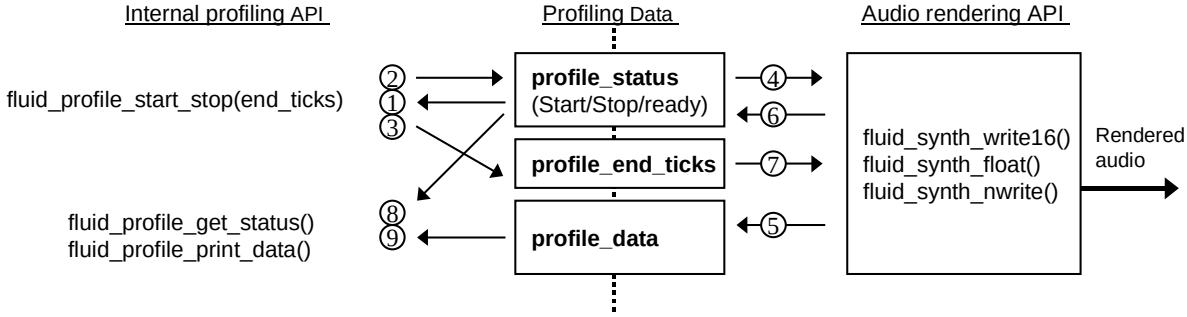

Fig.3.2 shows, internal communication variables between Internal profiling API and audio rendering API The protocol is the following::

profile_status: *request* measurement and result *status*. The state are:

- Initial state is PROFILE_STOP, "audio rendering API" doesn't not collect data.
- With duration > 0, *profile_status* is set to PROFILE_START by **fluid_profile_start_stop()**(2) if a measure isn't already running (1). This is a request to "audio rendering API"(4) to collect data in *profile_data* (5).
  If a measure is already running (PROFILE_START) (1), **fluid_profile_start_stop()** does nothing.
- Sets to PROFILE_READY (6) by the "audio rendering API" signaling to **fluid_profile_get_status()** (8) that data are ready, and signaling the "audio rendering API" (i.e itself) that data collect must stop (4).
- With duration à 0, profile_status is set to PROFILE_STOP (2) by **fluid_profile_start_stop()** to request the "audio rendering API" to stop data collect (4) in *profile_data* (5).

profile_end_tick: the end position of data collect in tick

- sets by **fluid_profile_start_stop()** (3) when starting a measure (PROFILE_START (2)) to pass to the "audio rendering API" (7) the position at which the collect must end.

- During the collect, the "audio rendering API" checks if the current position (*tick_since_start*) reachs *profile_end_tick* position. In this case, the API sets profile_status to PROFILE_READY.

profile_data: data collect
- Data are cleared by **fluid_profile_start_stop()**() before starting a measure (PROFILE_START) (2)
- Data are collected by audio rendering API (5) when a measure is running (PROFILE_START) (2)
- Data are read and displayed by **fluid_profile_print_data()** (9) when they are ready (PROFILE_READY) (8).

Following variables are default parameters useful only by **prof_start** command**:**
- ***profile_notes, profile_bank, profile_prog***: *notes* number, *bank* and *prog* preset numbers set by **prof_set_notes** command
- ***profile_print***, print mode set by **prof_set_print** command.
- ***profile_n_prof, duration*.** mesures number and duration of a measure set by **prof_start** command.
- ***profile_lock*,** mutual exclusion between possible multiple shell (see 3.2.3).

Inside fluid_sys.h enabled by WITH_PROFILING set to 1
#if WITH_PROFILING

/* "prof_start" shell command default parameters in fluid_sys.c */
extern unsigned short **fluid_profile_notes**; */* number of generated notes */*
extern unsigned char **fluid_profile_bank**; */* bank,prog preset used by */*
extern unsigned char **fluid_profile_prog**; */* generated notes */*

extern unsigned char **fluid_profile_print**; /* print mode */

extern unsigned short **fluid_profile_n_prof**; */* number of measures */*
extern unsigned short **fluid_profile_dur**; */* measure duration in ms */*
extern int **fluid_profile_lock** ; */* lock between multiple shell */*

```
/*-----------------------------------------------
  Internal profiling API (in fluid_sys.c)
-----------------------------------------------*/
/* Start a profiling measure used in shell command "prof_start" */
void fluid_profile_start_stop(unsigned int end_ticks, short clear_data)
 /* print profiling data used in shell command "prof_start" */
int fluid_profile_get_status(void);
void fluid_profiling_print_data(double sample_rate,fluid_ostream_t out);
/* logging profiling data (used on FluidSynth instance deletion)  */
void fluid_profiling_print(void);

/* Returns True if profiling cancellation has been requested */
int fluid_profile_is_cancel_req(void);

/*-----------------------------------------------
  Profiling Data (in fluid_sys.c)
-----------------------------------------------*/
/** Profiling data. Keep track of min/avg/max values to execute a
   piece of code. */
typedef struct _fluid_profile_data_t
{
  int num;
  char* description;          /* name of the piece of code under profiling */
  double min, max, total;     /* duration (microsecond) */
  unsigned int count;         /* total count */
  unsigned int n_voices;      /* voices number */
  unsigned int n_samples;     /* audio samples numbers */
```

```
} fluid_profile_data_t;


enum
{
        /* commands/status  (profiling interface) */
        PROFILE_STOP,     /* command to stop a profiling measure */
        PROFILE_START,    /* command to start a profile measure */
        PROFILE_READY     /* status  to signal a profiling measure has finished and
                             ready to be printed */
        /*- State returned by fluid_profile_print_if_ready() -*/
        /* between profiling commands and internal profiling API */
        PROFILE_RUNNING,  /* a profiling measure is running */
        PROFILE_CANCELED, /*  a profiling measure has been canceled */
 };

/* Data interface */
extern unsigned char fluid_profile_status ;      /* command and status */
extern unsigned int fluid_profile_end_ticks;     /* ending position (in ticks) */
extern fluid_profile_data_t fluid_profile_data[]; /* Profiling data */

/*-----------------------------------------------
  Macros
-----------------------------------------------*/
/** Macro to collect data, called from internal functions inside audio
   rendering API */
#define fluid_profile(_num,_ref,voices,samples) \
{ \
        if ( fluid_profile_status == PROFILE_START) \
        { \
                double _now = fluid_utime(); \
                double _delta = _now - _ref; \
                fluid_profile_data[_num].min = _delta < fluid_profile_data[_num].min ? \
                        _delta :\
                        fluid_profile_data[_num].min; \
                fluid_profile_data[_num].max = _delta > fluid_profile_data[_num].max ? \
                        _delta :\
                        fluid_profile_data[_num].max; \
                fluid_profile_data[_num].total += _delta; \
                fluid_profile_data[_num].count++; \
                fluid_profile_data[_num].n_voices += voices;\
                fluid_profile_data[_num].n_samples += samples;\
                _ref = _now; \
        } \
}

/** Macro to collect data, called from audio rendering API (fluid_write_xxxx()).
 This macro control profiling ending position (in ticks)
*/
#define fluid_profile_write(_num,_ref, voices, samples) \
{ \
        if (fluid_profile_status == PROFILE_START) \
        { \
                if (fluid_synth_get_ticks(synth) >= fluid_profile_end_ticks) \
                { \
                        /* profiling is finished */  \
                        fluid_profile_status = PROFILE_READY;\
```

```
                } \
                else \
                { /* acquire data */ \
                        double _now = fluid_utime(); \
                        double _delta = _now - _ref; \
                        fluid_profile_data[_num].min = _delta < fluid_profile_data[_num].min ? \
                                        _delta : fluid_profile_data[_num].min; \
                        fluid_profile_data[_num].max = _delta > fluid_profile_data[_num].max ? \
                                        _delta : fluid_profile_data[_num].max; \
                        fluid_profile_data[_num].total += _delta; \
                        fluid_profile_data[_num].count++; \
                        fluid_profile_data[_num].n_voices += voices;\
                        fluid_profile_data[_num].n_samples += samples;\
                        _ref = _now; \
                } \
        } \
}

#else
/* No profiling */
…….
…….
#define fluid_profile(_num,_ref, voices, samples)
#define fluid_profile_write(_num,_ref, voices, samples)

#endif /* WITH_PROFILING */
```

3.2.3. Remark: muti-task access considerations

We remark that profiling measurement is only useful when the profile API is called by only one shell task at a time.
For this reason there is not exclusive acces protection used inside Profiling interface API function (fluid_profile_start_stop(), fluid_profile_get_status())

However, using the console application, there is only one shell (by default). But we can start a server which allows multiple shell from remote consoles. In this case , the "profile" command can be executed by mutiples shell at the same time. To avoid this situation, a lock variable is used (profile_lock) . A simple flag with atomic acces protection is enough.

Thus the 3 following interface variables are assumed accessed by the "profile internal API "in the context of only one shell task, and by the "audio rendering API" in the context of only one audio task.
The communication protocol is that described in chapter 3.2.2, we notes that:

- ***profile_status*** variable is a mutual synchronization between the API profile (writting) and the audio rendering API audio (reading) or vice versa. As the variable is a byte only accessed by this 2 task and only one at a time, access is not critical.
- ***profile_end_tick*** variable is only written by profile API et only read by audio rendering API l'API audio_rendering. writting and reading access are synchronized by profile_status and are never simultaneous. So, access is not critical.
- ***profile_data*** variable is read and written by both API but access are never simultaneous (synchronized by profile_status). So, access is not critical

Conclusion:
1) As there are only one shell task and only one audio task
2) As the communication protocol is based on mutual synchronization
These variables doesn't need exclusive access protection .

3.2.4. Commands integration in the default commands set **(fluid_cmd.c, .h)**

Those four "profile" commands are added in the default commands set **fluid_commands[]**.
In fluid_cmd.c, commands existence is valided by WITH_PROFILING macro set to 1.

In fluid_cmd.c

```
#if WITH_PROFILING
/* Profiling-related commands */
{  "profile", "profile", (fluid_cmd_func_t) fluid_handle_profile, NULL,
    "profile                    Prints default parameters used by prof_start"},
{ "prof_set_notes", "profile", (fluid_cmd_func_t) fluid_handle_prof_set_notes, NULL,
    "prof_set_notes nbr [bank prog] Sets notes number generated by prof_start"},
{ "prof_set_print", "profile", (fluid_cmd_func_t) fluid_handle_prof_set_print, NULL,
    "prof_set_print  mode          Sets print mode (0:simple, 1:full infos)"},
{ "prof_start", "profile", (fluid_cmd_func_t) fluid_handle_prof_start, NULL,
    "prof_start [n_prof [dur]]       Starts n_prof measures of duration(ms) each"},
#endif
```

3.2.5. Implementing command: **profile (fluid_cmd.c)**

The command displays defaults parameters used by prof_start command
Default parameters are changed by the others "profiling" commands:.

- **profile_notes**: number of notes generated automatically.
- **profile_bank**, profile_**prog**: bank an prog preset numbers.
- **profile_n_prof**: numbers of measure.
- **profile_dur**: measure duration.

```
/*
handlers: profile
        Print default parameters used by prof_start
        Notes:0, bank:0, prog:16, print:0, n_prof:1, dur:500 ms
*/
int
fluid_handle_profile(fluid_synth_t* synth, int ac, char** av, fluid_ostream_t out)
{
}
```

3.2.6. Implementing command: **prof_set_notes (fluid_cmd.c)**

The command **prof_set_notes nbr [bank,prog]** allows to choose the number of notes that will be generated by the **prof_start** command before starting a burst of measures (3.2.8).
bank prog parameters are optionals. When there are given they change the default values.

- ***nbr*** is the number of notes (0 by default). When 0, no notes will be generated.
- ***bank*** et ***num*** are bank (0 to127) and prog(0 to 127)) preset number in the soundfont.

```
/*
        handlers: prof_set_notes  nbr  [bank prog]
        nbr: notes numbers (generated on command "prof_start").
        bank, prog: preset bank and program number (default value if not specified)
 */
int
fluid_handle_prof_set_notes(fluid_synth_t* synth, int ac, char** av, fluid_ostream_t out)
{
}
```

3.2.7. Implementing command: **prof_set_print (fluid_cmd.c)**

The command **prof_set_print mode** allows to choose print mode used by prof_start (see 3.2.14)

- mode 0 (simple display) or 1 (full display)

```
/*
        handlers: prof_set_print  mode
        mode: result print mode(used by prof_start").
                0: simple printing, >0: full printing
*/
int
fluid_handle_prof_set_print(fluid_synth_t* synth, int ac, char** av, fluid_ostream_t out)
{
}
```

3.2.8. Implementing command **prof_start (fluid_cmd.c)**

The user starts a burst of measure using this command: **prof_start [n_prof [dur]].**
n_prof, dur parameters are optionals. When there are given they change the default values.

- ***n_prof*** and ***dur*** in ms are the number of measures and the width duration of one mesure .

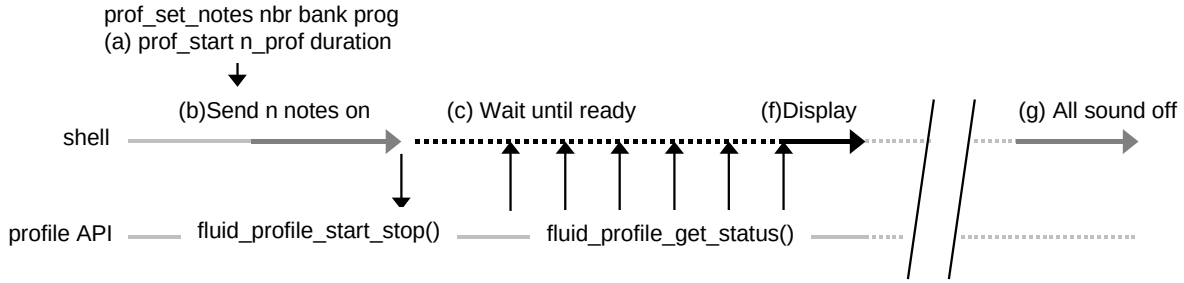

Fig.2

The command executes following steps (see Fig.2)

- (b) eventually generate simultaneaous notes : **fluid_profile_send_notes()** (see 3.2.9)
- for each mesure *iProf*
  - triggering measure *iProf* until end_ticks: **fluid_profile_start_stop(end_tick)** (see 3.2.11).
  -(f) passive synchronization on wainting results (see note 2).
- Stopping generated voices (see 3.2.10).

note 2: during this step waiting is passive **fluid_profile_get_status()** (3.2.13) is used.

```
/*
handlers: prof_start  [n_prof   [dur] ]
        n_prof  number of measure (default value if not specified).
        dur: measure duration (ms) (defaut value if not specified).
 */
int
fluid_handle_prof_start(fluid_synth_t* synth, int ac, char** av, fluid_ostream_t out)
{
}
```

3.2.9. notes generation: **fluid_profile_send_notes()(fluid_cmd.c)**

To generate simultaneous notes, the notes are played on different key number starting from MIDI channel 0 to 15.
The preset number ***profile_bank*** and ***profile_prog*** is used.
Velocity is limited to 30

```
/* Generate simultaneous notes for precise profiling
```

synth, synthesizer instance
notes, the number of notes to generate
bank, ***prog***, preset number used
out, output device
Returns the number of voices generated. It can be lower that the number of notes generated when the preset have instrument only on certain key range.
*/

void **fluid_profile_send_notes**(fluid_synth_t* synth, int notes, int bank, int prog, fluid_ostream_t out))
{
}

3.2.10. Stopping generated voices

Steps are:
- reset
- wait until all voices become inactives. This step is necessary to be sure that no voice is playing before restarting a new burst of measures.

3.2.11. Profile API start/stop a measure: **fluid_profile_start_stop() (fluid_sys.c)**

In fluid_sys.c, the existance of API is valided by WITH_PROFILING macro set to 1

/**
* Starts or stops profiling measurement.
* The function is an internal profiling API between the "profile" command
* prof_start and audio rendering API (see FluidProfile.pdf - 2.4.2).
*
* @param **end_tick** end position of the measure (in ticks).
* - If end_tick is greater then 0, the function starts a measure if a measure
* isn't running. If a measure is already running, the function does nothing
* and returns.
* - If end_tick is 0, the function stops a measure.
* @param **clear_data**,
* - If clear_data is 0, the function clears fluid_profile_data before starting
* a measure, otherwise, the data from the started measure will be accumulated
* within fluid_profile_data.
*/
This API follows the communication protocol described in 3.2.2.

- This Profile API est is used by **prof_start** (see 3.2.8) to start a measure.

/* Internal profile API */
void fluid_profile_start_stop(unsigned int end_ticks, short clear_data)
{
}

3.2.12. Cancelling a profiling: **fluid_profile_is_cancel_req() (fluid_sys.c)**

Returns true if the user asks to cancel the current profiling measurement.
Actually this is implemented using the <cr> key.

To implement this functionnality on an OS the macro FLUID_PROFILE_CANCEL must be defined.

1) Adds #define **FLUID_PROFILE_CANCEL** in fluid_sys.h.
2) Adds the necessary code inside **fluid_profile_is_cancel_req().**

Actually the function is implemented for Windows and linux.

3.2.13. Profile API display results: **fluid_profile_get_status(fluid_sys.c)**

In fluid_sys.c, the existence of API is valided by WITH_PROFILING macro set to 1

```
/**
* Returns status used in shell command "prof_start".
* The function is an internal profiling API between the "profile" command
* prof_start and audio rendering API (see FluidProfile.pdf - 2.4.2).
*
* @return status
* - PROFILE_READY profiling data are ready, the function prints the result.
* - PROFILE_RUNNING, profiling data are still under acquisition.
* - PROFILE_CANCELED, acquisition has been cancelled by the user.
* - PROFILE_STOP, no acquisition in progress.
*
* When status is PROFILE_RUNNING, the caller can do passive waiting, or other
* work before recalling the function later.
*/

/* Internal profile API */
int fluid_profile_get_status(void)
{

}
```

3.2.14. Printing data profiling: **fluid_profile_print_data() (fluid_sys.c)**

The function print the data in fluid_profile_data

```
/* print profiling data (used by profile shell command: prof_start)
* @param sample_rate sample rate of audio output.
* @param out output stream device
*/
void fluid_profiling_print_data(double sample_rate, fluid_ostream_t out)
{
        if (fluid_profile_print)
        {
                /* print alls details */
        }
        / * print cpu load */
}
```

The function print result using the print mode **fluid_profile_print** choosen by the command **prof_set_print** (3.2.7).

- when print_mode is >0, the function prints details (duration in µs) (see 3.1.2 Fig.4).
- when print_mode est 0, the function print cp load only (**fluid_profiling_print_load()**). Data collected allows the printing specified in 3.1.2 Fig.3.

Cpu load depends on following data:

- ***total*** : mesure duration (in µs).
- ***n_samples***: numbers of samples collected.
- **sample_rate:** audio sample rate.

load(%) = 100 x ((total / n_samples) / (1000000 / sample_rate))

- load(%) = (total x sample_rate) / (n_samples x 10000)

- **load(%) = (total x sample_rate) / (n_samples x 10000.0)**

n_samples is a required data in **fluid_profile_data_t**

3.2.15. Macros to collect data by audio rendering API**(fluid_sys.h)**

As explained in 2.2.3, data are collected in ***fluid_profile_data[]*** by audio rendering API **fluid_synth_nwrite_float()** ou **fluid_synth_write_float()** or **fluid_synth_write_s16()** each time this API is called. The inner audio API functions (inside fluid_synth_write_xxx()) collect data also.

Both macros **fluid_profile_ref_var(_ref), fluid_profile(_num,_ref,voices, samples)** (in fluid_sys.h) , allows the collect.
However only the "measure point" inside the API (not thoses in the inner function) controls the collect ending in all "measure points" (these of the fluid_synth_write_xxx() API and those of inner functions).

Thus, it is necessary to have a different macro for the point measure in the audio rendering API.
This macro **fluid_profile_write()** follows the communication protocol defined in 3.2.2, marked in bold.

```
#define fluid_profile_write(_num,_ref, voices, samples) \
{ \
        if (fluid_profile_status == PROFILE_START) \
        { \
                if (fluid_synth_get_ticks(synth) >= fluid_profile_end_ticks) \
                { \
                        /* profiling is finished */  \
                        fluid_profile_status = PROFILE_READY;\
                } \
                else \
                { /* acquire data */ \
                        double _now = fluid_utime(); \
                        double _delta = _now - _ref; \
                        fluid_profile_data[_num].min = _delta < fluid_profile_data[_num].min ? \
                                        _delta : fluid_profile_data[_num].min; \
                        fluid_profile_data[_num].max = _delta > fluid_profile_data[_num].max ? \
                                        _delta : fluid_profile_data[_num].max; \
                        fluid_profile_data[_num].total += _delta; \
                        fluid_profile_data[_num].count++; \
                        fluid_profile_data[_num].n_voices += voices;\
                        fluid_profile_data[_num].n_samples += samples;\
                        _ref = _now; \
                } \
        } \
}
```

The macro **fluid_profile()** is used by inner audio functions
This macro **fluid_profile(_num,_ref, voices, samples)** follows the communication protocol defined in 3.2.2, marked in bold .

```
#define fluid_profile(_num,_ref,voices,samples) \
{ \
        if ( fluid_profile_status == PROFILE_START) \
        { \
                double _now = fluid_utime(); \
                double _delta = _now - _ref; \
                fluid_profile_data[_num].min = _delta < fluid_profile_data[_num].min ? \
                        _delta :\
                        fluid_profile_data[_num].min; \
```

```
                fluid_profile_data[_num].max = _delta > fluid_profile_data[_num].max ? \
                        _delta :\
                        fluid_profile_data[_num].max; \
                fluid_profile_data[_num].total += _delta; \
                fluid_profile_data[_num].count++; \
                fluid_profile_data[_num].n_voices += voices;\
                fluid_profile_data[_num].n_samples += samples;\
                _ref = _now; \
        } \
}
```

3.3. How to apply patch: 0004-fluid_profile.path to v2.0

This chapter describes how to apply "profile" patch **0004- fluid_profile-to-v2.0.patch**

List of files concerned
Files
fluid_sys.h
fluid_sys.c
fluid_synth.c
fluid_voice.c
fluid_rvoice_mixer.c
fluid_cmd.c
fluid_cmd.h

- Note that the patch is added only in Fluidsynth library. Console application is not changed. To add commands profiling functionality , " 3 steps are necessary:

1) Applying profiling patch : **0001-profiling-0004-for-v2.0.patch**

- put the file **0001-profiling-0004-for-v2.0.patch** into the parent directory of **fluidsynth** working directory
- from the **fluidsynth** working directory verify the presence of **0001-profiling-0004-for-v2.0.patch**.

```
/GitHub/fluidsynth (master)
$  ls ../*.patch
../ 0001-profiling-0004-for-v2.0.patch
```

- invoke **git apply**

```
/GitHub/fluidsynth (master)
$ git apply --verbose  ../0001-profiling-0004-for-v2.0.patch
```

2) Configure with enable-profiling option using cmake. (-D enable-profiling).
3) Build the library.

3.4. fluid_utime() precision - recommendations

Time measurement made by profiling probe (see 2.2.3, 3.2.15) are done with **fluid_utime()** function for an espected precision of 1 µs.
For profiling we need high precision clock given by **g_get_monotonic_time()** if available (glib version >= 2.53.3).
If glib version is too old and in the case of Windows the function uses high precision performance counter instead of g_getmonotic_time().

3.4.1. Recommendation – using hardware performance counter when possible

For intel harware plateform, hardware performance counter brings about 0,3 µs precision when driven

by a 3 Mhz clock frequency. Theses counter are by far away the best choice for performance measurement. Fortunately Glib **g_get_monotonic_time()** (version >= 2.53.3) and OS Windows offers acces API to this counter. See fluid_utime() in fluid_sys.c.

3.4.2. Recommendation – using high audio.period-size

When it is not possible to use Intel precision hardware counter, there is a way to diminish the lack of **fluid_utime()** precision. It is **hightly recommended** to augment audio buffer size (setting audio.period-size (> 512) (i.e 4096...) to set a high latency (i.e 1 second).

Effectively, increasing size of audio buffers, increases audio rendering API duration and reduces imprecision.

3.5. Results - List of hardware

This chapter is a list of hardware measurement

3.5.1. HP Vectra VL 420 MT - Pentium(R) 4 CPU 1.70 GHz (CPU: 1 core)

Using performances counter: QueryPerformanceFrequency(),QueryPerformanceCounter()

Notes nbr	audio.period-size	cores	total load(%)	estimated maxVoices
200	256	1	98	218
200	512	1	94	226
200	1024	1	91	233
200	2048	1	88	241
200	4096	1	88	243

Using glib g_get_current_time() that use GetSystemTimeAsFileTime()

Notes nbr	audio.period-size	total load(%)	estimated maxVoices
200	4096	1	82	240

3.5.2. Board Gigabyte GA-MA785GM-US2H F5 - CPU AMD Phenom™ || x4 955

CPU: 1 core

Using performances counter: QueryPerformanceFrequency(),QueryPerformanceCounter()

Notes nbr	audio.period-size	cores	total load(%)	estimated maxVoices
200	256	1	34.82	611
200	512	1	28.47	745
200	1024	1	25.65	830
200	2048	1	24.05	888
200	4096	1	23.41	911

CPU: Using multi-cores

Using performances counter: QueryPerformanceFrequency(),QueryPerformanceCounter()

Notes nbr	audio.period-size	cores	total load(%)	estimated maxVoices
200	512	1	28.47	745
200	512	2	15.21	1491
200	512	3	10.39	2319
200	512	4	8.38	3015

3.5.3. Board D845 GERG2 / D845 PECE - Pentium(R) 4 CPU 2.40 GHz (CPU 1 core)

Using performances counter: QueryPerformanceFrequency(),QueryPerformanceCounter()

Notes nbr	audio.period-size	Rev -Chor	total load(%)	estimated maxVoices
200	512	On-On	63.5	336
320	512	On-On	99	334
320	512	Off-Off	91	350