emufxtool

Author: Nicola Orru' <nigu@itadinanta.it>

23rd March 2003

Abstract

This project aims at supplying a sort of emu10k1-swiss-army-knife for the ALSA project. The core of emufxtool is an assemble/unassemble engine that can convert the program and data contained in the DSP memory in and an human-readable form and vice-versa.

The emu10k1 DSP chip is the core of the creative SB Live!, Audigy, PCI 512 and other sound cards.

Version: 0.2

Parent-Project: ALSA

Based-on: as10k1 A0.99

Keywords: Linux, Sound, ALSA, Creative, emu10k1, fx8010, SoundBlaster, assembler, effects

Original-Author: Daniel Bertrand <d.bertrand@ieee.ca>, Jaroslav Kysela <perex@suse.cz>.

1 Project History

I know I ought not start this document from the history of a project that has no history, yet. You may ask why. Or you may not. However, you'll find an answer at the end of this section.

This project got life the minute I felt the need for chorus and reverb[1] to be enabled in my SoundBlaster Live! WaveTable midi synthesiser, just like the Creative Driver does under Windows 2000, so I subscribed immediately to alsa-devel and asked the list about what to do to join in.

Someone (maybe Takashi Iwai or Jaroslav Kysela, I don't remember -Yes, I know I should peek at the archives, but let me tell the rest of the story-) wrote me in response that I (or anyone else) had to sew a patch to the kernel driver in order to enable the FX_BUSes for chorus and reverb, and to write the loader code, while an assembler program that appeared to be suitable for ALSA had already been provided by **Daniel Bertrand** and **Jaroslav Kysela** (thanks!).

This is the way my odyssey started: I downloaded all the docs I was able to find about the DSP, the assembler code, the driver source, then I began to try and experiment with them... just to discover that a loader based on an OSS-oriented assembler couldn't work, first of all because of the completely different patch management mechanisms and effect routing inside the drivers. So I had to start again. Fortunately, most of the code that I wrote to perform microcode unassembly was useful, thus it was immediately embedded in the new version of the tool.

Trying to expand my horizons, I moved the project's objective, from a simple loader to the aforementioned "swiss knife" for the fx8010 DSP processor. What was originally thought as "ld10k1" (loader for emu10k1) became "emufxtool".

That's where I am: I managed to implement a beta version of a tool that can

- assemble DSP code, featuring a powerful one-pass macro preprocessor, an expression parser/evaluator, and typed variable declarations

[1]By the way, the tool still CAN'T enable chorus and midi reverb : (

- unassemble contents of the DSP registers and code in an human-readable format, that can be easily modified, recompiled and reloaded.
- unassemble compiled DSP patches
- load compiled DSP patches into the emu10k1 iron (with a relocation mechanism, designed but still to be implemented)

and, meantime, there's where I got stuck. My comprehension of how the fx8010 (the core of the emu10k1) works is far from complete, neither my knowledge of ALSA internals is. I have to go and study some more (and more), assuming I will be able to obtain more documentation about the chip.

Now you have the answer to the initial question: I need help. If you reached this very word, maybe you're interested in giving me a hand. If not... Well, who am I supposed to be speaking to? :)

Anyway, what I need the most is feedback from experts in the ALSA architecture and, particularly, in the emu10k1 driver. Feedback from testers and users is also vital.

2 Installation

2.1 Requirements

What you do need are:

- GCC 3.2
- the C++ standard template library (installed in /usr/include/c++/3.2
- ALSA 0.9.0rc7 (or later?)
- GNU make

Soon (or later) I will provide an autoconf/automake makefile generator and installer. Once I learn autoconf and automake...

2.2 Building

You can make and run this program by the canonical sequence:

```
tar -xvzf emufxtool-version.number.tar.gz
cd emufxtool-version.number
make
make install (as root)
make test (as root)
```

If make terminates successfully, you will end up with a

```
emufxtool
```

binary executable in /usr/local/bin.

3 Usage

3.1 Invocation

```
emufxtool action [-o asmoptions] [-d device, defaults to hw:0,0] \
[-y additional_sYmbol_table] [dsp patch files...]
```

action

'-l'	load patches into emu hardware
'-t'	test patches sanity, useful with -oa, see later
'-a'	assemble patches
'-h'	help
'-v'	version
'-u'	unassemble patch found in emu hardware
'-s'	save hardware contents in a patch file

asmoptions (multiple options allowed, latest override earliest):

'a'	all options on (default all off)
'v'	enables
'V'	disables patch dump to stdout
't'	enables
'T'	disables token analysis dump to stdout
'p'	enables
'P'	disables parsing only (no files generated during compilation)
'm'	enables
'M'	disables macro expansion to stdout

Notes

- Every compilation process generates a "patch_name.emufx" file for each assembled patch, overriding any previously existing file named that way.
- Every compilation process generates an additional "patch_name.sym" containing the symbols, that can be used later to unassemble the same patch or the DSP contents properly, overriding existing files.

3.1.1 Examples

```
emufxtool -a default.as10k1
```

compiles "default.as10k1", creating the binary patch "default.emufx"

```
emufxtool -a -oa *.as10k1
```

compiles any patches found in the current path, creating a *.emufx for each file, dumping a very verbose list of messages to stdout

```
emufxtool -t -ov default.emufx -y reverse.sym
```

dumps all the contents of the binary patch default.emufx to stdout, using the symbols contained in reverse.sym

```
    emufxtool -l -d hw:0,0 default.emufx
```

loads default.emufx into the DSP memory and registers through the hw:0,0 interface

```
    emufxtool -h
```

help

```
    emufxtool -u -ov
```

dumps the contents of the DSP memory onto stdout

```
    emufxtool -s dump.emufx
```

dumps the contents of the DSP memory to a binary patch file called dump.emufx

4 Language Syntax

4.1 Meta-Language

The language structure is described by the following BNF-like rules, where

"string"	is a fixed literal string
'expr'	is a regular expression
{}	means zero or more items,
[]	means zero or one item
::=	means "means"
<null>	represent an empty string, i.e. nothing
bare-words	represent language elements

4.2 Basic Rules

The language is "free form", meaning you can insert any number of spaces, tabs and newlines between elements as you prefer (except inside strings).

The assembler applies no limits to line length or word length. All identifiers and keywords are case sensitive. The syntax was chosen with an eye to the preservation of compatibility with the original as10k1 project, in order to ease the porting of existing OSS patches to ALSA.

4.2.1 Patch structure

A so called "assembly patch" must be written in an ASCII text file. In a patch, you can mix, as you like, instructions and declarations. You can use PREPROCESSOR DIRECTIVES whenever you like, for both instructions and declarations. Preprocessor directives will be described aside, as they don't represent language statements.

patch	::= {element [";"]}
element	::= instruction | declaration | <null>

4.2.2 instructions

All instructions require 4 operands. The format is:

instruction	::= [label] opcode expression "," expression "," expression "," expression [";"] | [label] <null>
label	::= identifier":"
opcode	::= "MACS" | "MACS1" | "MACW" | "MACW1" | "MACINTS" | "MACINTW" | "ACC3" | "MACMV" | "ANDXOR" | "TSTNEG" | "LIMIT" | "LIMIT1" | "LOG" | "EXP" | "INTERP" | "MAC0" | "MAC1" | "MAC2" | "MAC3" | "MACNS" | "MACNW" | "MACINT" | "MAC-INTNW" | "LIMIT0" | "LIMITGE" | "LIMITL"

where expressions mean operands in the following order: R, A, X, Y. Note that some documentation out there call the R operand Z and the A operand W.

Later in this document you can see what an "expression" is.

Opcodes Here are 16 opcodes.

MACS:	R = A[2] + (X * Y >> 31) ; saturation
MACS1:	R = A + (-X * Y >> 31) ; saturation
MACW:	R = A + (X * Y >> 31) ; wraparound
MACW1:	R = A + (-X * Y >> 31) ; wraparound
MACINTS:	R = A + X * Y ; saturation
MACINTW:	R = A + X * Y ; wraparound (31-bit)
ACC3:	R = A + X + Y ; saturation
MACMV:	R = A, acc += X * Y >> 31
ANDXOR:	R = (A & X) ^ Y
TSTNEG:	R = (A >= Y) ? X : ~X
LIMIT:	R = (A >= Y) ? X : Y
LIMIT1:	R = (A < Y) ? X : Y
LOG:	... /* FIXME: how does it work? */
EXP:	... /* FIXME: how does it work? */
INTERP:	R = A + (X * (Y - A) >> 31) ; saturation
SKIP:	R,CCR,CC_TEST,COUNT

several of these have predefined aliases:

MACS	is aliased by MAC0
MACS1	is aliased by MAC1 or MACNS
MACW	is aliased by MAC2
MACW1	is aliased by MAC3 or MACNW

[2]Special note on the accumulator: MAC* instructions with ACCUM as A operand use Most significant 32 bits. MACINT* instructions with ACCUM as A operand use Least significant 32 bits.

MACINTS	is aliased by MACINT or MACINT0
MACINTW	is aliased by MACINT1
LIMIT	is aliased by LIMIT0 or LIMITGE
LIMIT1	is aliased by LIMITL

For more details on the emu10k1 see the dsp.txt file distributed with the linux driver.

You may optionally terminate an instruction by a ";". You can chain more than one instruction on the same line or subdivide one instruction into many lines.

Operands are always specified as expressions. The assembler evaluates the expression, then compiles the result into an address of a register (as the operands for emu10k1 instructions are always addresses of registers), according to the expression operands type.

expression ::= "(" expression ")" | expression operator expression | literal_constant | identifier | unary_operator expression | array_element | label_reference

operator ::= "+" | "-" | "^" | "<<" | ">>" | "==" | ">" | "<" | "<=" | ">=" | "|" | "&" | "||" | "&&" | "*" | "%" | "/"

unary_operator ::= ~ | ! | - | +

identifier ::= '[A-Za-z][A-Za-z0-9\.]+'

array_element ::= identifier "[" expression "]"

literal_constant ::= decimal_constant | hex_constants | binary_constant |

floating-point_constant | octal_constant | time_constant

label_reference ::= ":" identifier

decimal_constant ::= '[0-9]+'

hex_constant ::= '0x[0-9a-fA-F]+' | '\$[0-9a-fA-F]+'

octal_constant ::= '@[0-7]+'

binary_constant ::= '%[01]+'

floating-point_constant ::= '#?'decimal_constant'(\.decimal_constant)?''(E[+-]decimal_constant)?'

time_constant ::= "&"decimal_constant'(\.decimal_constant)?''(E[+-]decimal_constant)?'

Notes

- Floating point values are always (and immediately) converted to fixed point numbers.
- Time constants are expressed in seconds and immediately converted to samples (integer). So, for instance, &1.0 (one second) is converted to 48000; #1.0 is converted to 0x00010000.
- All arithmetic is performed in fixed point/integer space. If you are looking for examples, look at "default.as10k1"

4.2.3 Declarations

you can declare variables, symbolic constants, controls, TRAM lines, and TRAMs

Initialisers An initialiser is either a constant, a variable, a control, a TRAM or a line that is bound to an unique name. To declare one or more variables you should use the following syntax (refer to the expression syntax):

declaration ::= var_declaration | const_declaration | control_declaration | tram_declaration | tram_line_declaration ";"

var_declaration ::= "var" ["absolute"] ["export"] [var_type] var_initialiser {, var_initialiser}

const_declaration ::= "const" ["export"] var_initialiser {, var_initialiser}

control_declaration ::= "control" [control_type] [control_value] atom_initialiser {,atom_initialiser}

tram_line_declaration ::= "line" atom_initialiser {,atom_initialiser}

tram_declaration ::= "tram" identifier attribute_list

control_type ::= <null> | "mono" | "stereo"

control_value ::= <null> | "onoff" | "range"

var_initialiser ::= atom_initialiser | array_initialiser

var_type ::= <null> |"register" | "fxbus" | "extin" | "extout"

atom_initialiser ::= identifier ["=>" hex_constant] ["=" expression] [attribute_list];

array_initialiser ::= identifier ["[" expression "]"] ["=>" hex_constant] ["=" expression_list]

expression_list ::= "{" expression {,expression} "}"

attribute_list ::= "{" [attribute_definition] [,attribute_definition] "}"

attribute_definition ::= identifier "=" expression | identifier "=" string

string ::= "\"" '[^\"]*' "\""

Semantic

const When a CONSTANT is declared, using "const", it is inserted in a temporary symbol table, and its value used in expression. A GPR is allocated only if necessary; multiple constants can share the same GPR or use an hardware constant.

var When a VAR is declared, by means of "var", a GPR is allocated for each element if the var is not assigned a GPR directly. If a var is givenone or more values, they can be used as constants during assembly times. Arrays are always allocated contiguously. GPRs are assigned in declaration order (earlier declared vars are given the lowest addresses), starting from the first GPR available (usually 0x100 for the Live!). Vars can be declared "absolute" and thus an address can be assigned to the variable. This way, no GPRs are allocated for the variable. An example follows:

```
var absolute VARIABLE_NAME => 0x1ee;
```

Vars and constants always have empty attribute lists.

control When a "control" is declared, it can be either MONO or STEREO. Mono is a single value, whilst stereo means a vector (array) of two distinct values, to be used for left and right channels. This means a MONO control uses a GPR, and stereo control uses two GPRs. Valid attributes for controls in an attribute list are:

- "name" (string): contains the name of the control

- "min" and "max" (numeric): define the control's range
- "index" (numeric): assigns an internal index to the control (FIXME: let me understand this first!)
- "translation" (string): can be "t100" (default for "range" controls),
- "bass", "treble" or "onoff" (default for "onoff" controls). Sets the translation table inside the driver (FIXME: understand this, too...)

when a "stereo" control is declared, for instance as VOLUME, three variables are actually created:

VOLUME, that is an array of two elements, VOLUME[0] and VOLUME[1]

VOLUME.left, that aliases VOLUME[0]

VOLUME.right, that aliases VOLUME[1]

The control initial value can be assigned by "=" in the declaration

tram declarations define the amount of TRAM memory to allocate, for being used by "line" declaration.

line A "line" is meant to be a "point" where you can access tram memory, for reading and/or writing. The emu10k1 DSP reserves a different address space to lines, which is used to assign address to these variables. When you declare a "line" initialiser, actually three variables are created:

LINE, that points to the first "data line register" available

LINE.data that aliases "LINE"

LINE.address

Examples

```
control stereo range VOLUME = {100,100} {name="Main Volume", min = 0, max = 100};
```

declares a GPR stereo control called "Main Volume", with an initial value of 100, with range between 0 and 100

```
control stereo onoff VOLUME_SWITCH = 1 {name="Main Switch", index=0};
```

declares a GPR switch-type volume called "Main Switch", initially on

4.3 Macro preprocessor

The so-called preprocessor is not actually a preprocessor, as it works in "real time" during compilation with no need for a second-pass, but it performs the same way: you can write "parametric" constructs that can "expand" to actual code, preventing you from rewriting entire blocks that are alike. In a system like the emu10k1 DSP, where you can't use calls or loops, a macro system is the only way to emulate subroutines and structures.

But I guess that you already know what a preprocessor is, so let's go on.

Macro preprocessing allows you to write comments in "c-style" **/* */**. When a comment is encountered, its contents are unconditionally discarded. Nested comments are not allowed.

Macro blocks may contain everything, including other preprocessing directives, that should expand correctly. This behaviour lets you compose quite weird structures, like nested loops or even recursive macros (beware of infinite recursion!).

The directives start with a dotted ".keyword" and can be used everywhere in the code, being possible to declare them everywhere in the middle of a line, too.

```
.include "filename"
```

This directive substitutes itself with the contents of the included "filename"

```
.define macro_name(param1, param2...) { block }
.macro macro_name(param1, param2...) { block }
```

creates a macro called "macro name".

It substitutes itself with a null string, but whenever the defined symbol is found in the code, from now on, it will be expanded to "block". Parameter substitution is performed inside the "block". That is, if the macro is invoked as

```
macro_name(a,b,...)
```

the invocation is substituted with "block". Whenever "param1" is found in the block (as a distinct token) it is substituted by "a", param2 is replaced by "b" and so on. The number of arguments during invocation must be equal to the number of parameters the macro has been declared with.

Parameter-less macros can be declared (and invoked) as

```
macro_no_params()
```

with a null list of parameters. Parentheses are always mandatory (but this may change in the future). An alternative way to invoke a macro and perform substitution is:

```
macro_name a, b... ;
```

In this condition, the final ";" is mandatory, but this syntax is deprecated (although included) because it may lead to ambiguities.

```
.if expression { block }
```

This directive evaluates the expression using literal constant and initialisers defined before, and substitutes itself to "block" if the expression is not zero.

```
.if expression { block1 } else { block2 }
```

This directive evaluates the expression using literal constant and initialisers defined before, and substitutes itself to "block1" if the expression is not zero or "block2" if the expression evaluates to zero.

```
.eval expression
```

evaluates "expression" and substitutes itself to the result. Useful, in combination with ".define" if you have very complex constant expression often used in the code and you want to avoid re-evaluating them lots of times.

```
.defined symbol
```

replaces itself with "1" if the symbol is a defined initialiser or macro, or with "0" otherwise.

```
.undefined symbol
```

like the latter, with meaning inverted.

```
.for identifier = start:end { block }
```

start and end are expression that must evaluate to integer values and express a sequence (ascending or descending) of numbers. For each number in this sequence, the directive replaces itself with the "block", replacing the token "identifier" inside the block with the current value of the sequence number.

```
.file
```

replaces itself with the name of the file currently being read

```
.line
```

replaces itself with the line number currently being read

```
.warn { message }
```

replaces itself with nothing, expands "message" using the current macro definition and outputs it to stderr.

```
.err { message }
```

behaves like .warn, but generates an user error that leads to a failure of the compilation process. if a .err is encountered, the current compilation does not create an output file.

5 TODO

code style. At this stage, code actually works, although not perfectly, but not enough effort was spent to make the code readable, understandable or coherent.

documentation. No comment. The only existing documentation is this README. More documentation will be available as soon as possibile. I just started experimenting with doxygen.

TRAM setup support. Support for internal and external TRAM must be improved

PCM setup support. PCM setup is not supported yet

More sound cards. The only card tested and "fully" supported is the SoundBlaster Live! 256, as I haven't got other emu10k1 cards on my system. Other cards will be easily supported, as all card parameters are "modularized" in a single class.

automake/autoconf build system. I am not acquainted enough with autotools.

cvs support. As this project was intended to be a part of a largest one, I think the better choice I can take is delegating to ALSA maintainers the decision about where to store its files. They may be merged into the ALSA repository, so I guess it will be included in the "alsa-tools" cvs, or it survive as a standalone project. In the latter case, an autonomous repository will be set up.

6 How to contribute

You can help me in the development of this project by:

- Testing the program, submitting bug reports and wish lists
- Submitting patches or fragments of code
- Proofreading and reviewing code and documentation
- Sending me corrections and advices that can help me to improve my writing (english is not my first language)
- Any Other Business

6.1 Contact

You can contact me (Nicola Orru') for any reason, by email (of course), at the following addresses:

nigu@itadinanta.it (personal email)

no@energit.it (office email)

If you write me about this project, the email subject should start with the word **[emufxtool]**

Contents