Sun Feb 22, 2009 12:47 pm

Ok, here's another batch of info. When I conducted my previous tests on the envelope generator, I had to extrapolate the output from the envelope generator based on the analog output from the chip. This severely limited the accuracy and precision of the tests I could perform. Increased understanding of the phase generator, operator unit, and the test register, now allows me to obtain pure, digital samples of the output data from the envelope generator. With this new information, I decided a complete re-test of the envelope generator was required, as several clear errors had been identified, and there was a lot more testing I still wanted to carry out which I was unable to perform with sufficient accuracy before. I've now done a thorough re-test of the envelope generator as a whole, and as a result, I've got a lot of corrections for the information I provided before, as well as bunch of new information to share.

This post is going to cover details of the envelope generator, but unlike the information I posted previously, this is mainly going to be a list of corrections and additions to previous information I have provided. I am going to break it into points arbitrarily arranged and numbered, and it is going to primarily focus on what is different to my previous description of the envelope generator. Unless I specifically amend or correct a piece of information I provided in my previous description, the previous information still stands. The previous description I gave of the envelope generator is at the following location:
http://gendev.spritesmind.net/forum/vie ... =5716#5716

General info:
1. Envelope Generator update clock rate
I've posted this correction previously, but I'm going to mention it again briefly here. I previously stated that the envelope generator was updated according to the following clock rate:
"EG Clock = External Clock / 351"
Or that "The ratio of the frequency output to the envelope generator clock is 1:2.4375". This information was incorrect. In fact, the envelope generator is updated once every 3 fm update cycles, which was the existing behaviour in the MAME core. Refer to the following post for the information I previously gave on this correction:
http://gendev.spritesmind.net/forum/vie ... =6224#6224

2. Envelope generator output inversion
I previously stated that the envelope generator had an output inversion flag which was used by both SSG-EG mode and the attack phase. This is incorrect. SSG-EG has an output inversion flag, but it is not shared by the attack phase. The attack phase does not use output inversion, nor does any function of the envelope generator outside of SSG-EG mode.

3. Calculating the attack curve
The previously given information for calculating the attack curve stated that while the attack phase was active, the output inversion bit was set, and the following formula was used to advance the attack curve:

CODE: SELECT ALL

```
operator.attenuation += attenuationIncrement * (((1024 - operator.attenuation) / 16) + 1)
```

This is incorrect on both points. The attack phase does not use output inversion, and the provided update calculation is incorrect. The correct update formula is as follows:

CODE: SELECT ALL

```
newAttenuation += (~newAttenuation * attenuationIncrement) >> 4
```

This is the same as the existing update calculation used in the MAME core.

4. Restart of phase generator on key-on
I forgot to mention in my original post that when key-on occurs, the internal phase counter in the phase generator is reset to 0. This of course also includes key-on events triggered by CSM mode. The phase counter is also reset in SSG-EG mode at the beginning of each repetition of the waveform in cases where both the HLD and ALT bits are unset, but the implementation is a little different than you might expect. The SSG-EG section will cover this fully.

5. When register changes are applied
I didn't provide much information on this before, because I hadn't performed many tests on it before. I did state previously that the rate value for the current phase of the envelope generator was calculated and stored when entering each phase of the envelope, and that updates within each envelope phase were based on this stored value. This is incorrect. The rate value is calculated each update cycle of the envelope generator. As such, changes to the various rate parameters for an operator, as well as changes to the rate key-scale value, can cause an immediate change in the effective rate for the current phase of the ADSR envelope.

Regarding other register values, changes to all other parameters for an operator or channel, including TL, Detune, Multiple, Key Scale, AMS, PMS, Left/Right, Block, Fnum, etc, also all take effect immediately. One important note however relates to SL, the sustain level. Changes to SL do take effect immediately, including during the decay phase, but note that once the decay phase of the envelope reaches SL, the transition from decay to sustain is a one-way operation. If the ADSR envelope was in the decay phase, and the attenuation level reached SL, the ADSR envelope would switch to the sustain phase. If you were to then modify SL to move it past the current attenuation level (`eg`, so the operator was now in the sustain phase above SL), the operator does NOT re-enter the decay phase. All emulators probably do this correctly anyway, but it was possible that the decision of whether the operator was in the decay phase or the sustain phase was made each update cycle based on the relative position of the current attenuation value to the sustain level. This is not the case. Once the transition is made from the decay phase to the sustain phase, changes to SL will not have any effect.

Note that changes to the `SSG-EG` register also take effect immediately, but there's a lot to cover about this behaviour, which will be addressed separately in the `SSG-EG` section.

6. Clamping of attenuation values when changing states
I posted a comment somewhere around here about the possibility of the current attenuation level being "clamped" to SL when moving from the decay phase to the sustain phase. I suspected that if, for example, SL was 0x200, and the decay phase was to cause the current attenuation value to jump from, say, 0x1FC to 0x204, the attenuation value would be clamped to 0x200, preventing the attenuation level from moving past SL during the decay phase. This is incorrect, and the premise itself is also incorrect.

The sustain level can only be set in 16 increments of 0x20, from 0x0 to 0x1E0, with the special case that when it is set to max, it is forced to the maximum attenuation level (0x3FF). The decay phase always begins from an attenuation value of 0, since the attack phase does not terminate until this value is reached. Without `SSG-EG` mode active, the largest increment value which can be applied to the attenuation value in a single update cycle is 8. Since all the levels SL can reach are multiples of 8, without making changes to the effective rate of the decay phase during the decay sequence, the attenuation level would always finish perfectly on SL anyway, without `SSG-EG` active. With `SSG-EG`, the increment value is multiplied, however the 6x figure I gave previously is incorrect, as will be elaborated on below. The correct multiplier is 4x. With a 4x multiplier, the maximum increment value for a single step in the decay phase is 8*4=0x20, which is the same interval as the minimum step we can specify with SL, so again, unless changes are made to the effective rate of the decay phase during decay, we will always finish perfectly on SL, so no clamping is required. When we do change the decay rate during the decay phase, we have confirmed that the decay phase can be forced to "skip over" SL in both normal and `SSG-EG` modes. In the case where the decay phase moves past SL, the sustain phase begins on the next cycle, but the attenuation value is not clamped to SL.

7. When key on/off events are applied
In my implementation, I thought changes to the key-on state were applied according to the envelope generator update cycle, so only once every 3 fm update cycles would a key-on state change take effect. This is incorrect. In fact, key-on changes are processed every fm update cycle. This becomes important when CSM mode is in use, since timer A overflows every fm update cycle.

8. Attack phase with rate values 62 and 63
I was vague on this point before. I've got a bit more info to add now. When an attack phase has a rate value greater than or equal to 62 (0x3E or 0x3F), it is treated differently. With no special case handling, rate values of 62 and 63

would be the same as rate values 60 and 61, namely, there would be an 8x increment value applied to the attack curve update process, and from a starting attenuation level of 0x3FF, it would take 10 update steps to achieve minimum attenuation. In fact, for the attack phase, there is special case handling to force the attenuation level directly to 0 when the attack phase is entered, if the calculated rate value is 62 or 63. Note that this happens immediately when the key-on event is processed, so it doesn't even have to wait for the next full envelope generator update cycle. This behaviour also occurs when the SSG-EG envelope loops, so it isn't just actual key-on events which trigger this behaviour.

Since this special case handling appears to be triggered when the attack phase is entered, there's an interesting quirk that occurs when you change the attack rate during the attack phase. If you begin an attack phase with a slow rate value (to give you enough time to change it again before it finishes), then change the attack rate settings to switch to an effective rate value of 62 or 63 during the attack phase, the attack phase will NOT finish in the next update cycle. In fact, it won't advance at all. Switching to an effective rate value of 62 or 63 after the attack phase has already begun causes the attack phase to stall. The attenuation value will be locked at its current value, and will no longer advance until the attack rate is modified again to something less than 62, at which time, it will resume advancing from its current position, or until key-off occurs. I don't think MAME emulates this currently.

I haven't found a neat way to emulate all this behaviour. I currently calculate the effective attack rate immediately when switching to the attack phase, and if the rate is greater than or equal to 62, I force the attenuation level directly to 0. I also have an exclusion in the normal envelope generator update cycle when updating the attack curve. If the attack rate is greater than or equal to 62, I don't advance the attack curve. This emulates the behaviour of the YM2612, but it probably isn't emulating the exact way this special case behaviour is implemented. I'm open to any suggestions about a neater way to emulate this.

SSG-EG corrections:
The information I gave previously about SSG-EG mode is mostly correct, but there are a lot of critical implementation details which I didn't have at the time. These details are vital in order to emulate the multitude of quirks which can arise with SSG-EG mode when using an attack rate less than 0x1F, or when making changes to various register values, in particular the SSG-EG mode flags, during output. I'm going to cover corrections for clear errors in my previous description of SSG-EG in this section, and cover new information regarding SSG-EG in the following section.

1. SSG-EG decay phase update formula
I previously stated that during SSG-EG mode, for the decay, sustain, and release phases, the envelope generator attenuation value was updated as follows:

CODE: SELECT ALL

```
operator.attenuation += 6 * attenuationIncrement
```

This is incorrect. The increment value is in fact only multiplied by a factor of 4 during SSG-EG mode, making the correct update formula the following:

CODE: SELECT ALL

```
operator.attenuation += 4 * attenuationIncrement
```

This was the existing update behaviour in MAME, with the exception that MAME did not apply any multiplication factor in the release phase under SSG-EG mode. While MAME had the multiplication factor correct, this 4x multiplication factor should indeed be applied during the release phase under SSG-EG mode, as per the information I gave previously.

2. Attenuation limits under sustain, decay, and release
When I wrote my previous information, I believed that the decay, sustain, and release phases under SSG-EG decayed the entire way to the maximum attenuation level of 0x3FF. In particular, I believed that an SSG-EG pattern without the hold bit set would loop only when the attenuation level had progressed the entire length from 0x0 to 0x3FF. This is actually not the case. When the internal attenuation level reaches 0x200 under SSG-EG mode, a variety of special

operations are performed, which will be elaborated on later.

3. Output inversion
I mentioned previously that SSG-EG "inverts" the attenuation output, but I really didn't provide any information on exactly how the non-inverted data was converted to the inverted data, because I wasn't able to sample the digital output. The attenuation data is inverted on the output from the envelope generator through the following operation:

CODE: SELECT ALL

```
attenuation = 0x200 - attenuation;
attenuation &= 0x3FF;
```

Basically, it's a full, proper two's compliment inversion. The inversion is actually "centered" at 0x200. MAME does the output inversion by performing a simple XOR operation with 0x1FF, which is incorrect. This produces an output which is off by 1. The result needs to be adjusted for normal two's compliment math. 0x001 should invert to 0x1FF, 0x110 should invert to 0xF0, 0x250 should invert to 0x3B0, etc.

SSG-EG new info:
SSG-EG is dangerous to emulate. It's dangerous because many of its functions could be implemented in a dozen different ways to achieve the intended result. Consider the implementation of output inversion. Based on the information I previously gave describing how the inverted output works, if you were to think about emulating it, you would probably create a boolean flag, to indicate whether the output is currently inverted or not. In the case of the attack bit, you would probably use the attack bit to determine the initial state of this inversion flag, so when key-on occurs, you would load the value of the attack bit into the inversion flag. It seems simple and logical, it's what I did at first, and it's what MAME currently does. It's also wrong. Similar pitfalls exist with things like hold mode and the alternate bit.

Initially it may appear it doesn't matter how the YM2612 hardware performs a particular task compared with how it is emulated, as long as the two implementations produce an identical result. This is true, but unfortunately, there are a lot of ways to abuse SSG-EG mode. When you start changing things on the fly, and pushing SSG-EG into corners it's not supposed to deal with, you can expose lots of little quirks. These aren't bugs, simply areas of undefined behaviour where the result is not specified, because "you're not supposed to do that". These quirks betray information about how SSG-EG is really implemented. If your implementation differs from the YM2612 implementation, your SSG-EG implementation will not function correctly when pushing into these undefined areas. Unfortunately, simply using an attack rate less than 0x1F runs into a multitude of quirks, and we now know of several games which do in fact make use of this (Beavis & Butthead chainsaw sound effect, Barcelona Olympics water diving splash sound), so it's now important to emulate all these quirks, not just as an academic exercise, but in order to get correct sound in retail games.

The good news is, after a solid month of testing involving hundreds of tests on dozens of strange pieces of aberrant behaviour, I have managed to derive an accurate implementation of SSG-EG which fully implements all the quirks of the real hardware. This isn't a hackaround, it makes perfect sense, and the implementation is fairly straightforward in the end, but it is almost completely different to what I started with. MAME will also require its SSG-EG implementation to be rewritten from scratch, and significant modifications to the envelope generator implementation in general, in order to incorporate these changes, so it should keep someone busy. 

1. Where SSG-EG is updated: Envelope generator output cycle vs envelope generator update cycle
As previously mentioned, the envelope generator is updated once for every 3 times the phase generator is updated, and once for every 3 times the output of each operator is calculated. While the envelope generator may only update once every 3 FM output cycles however, its output is used every cycle. This means that internally, the envelope generator has a separate output process which runs at the same clock rate as the operator unit, to feed the current output for each operator to the operator unit as it is required. As it turns out, this output cycle is not an idle process. In fact, this output cycle is where virtually all the SSG-EG processing steps are implemented. Accurate emulation of SSG-EG mode requires that there be a series of active steps performed each time the output of the envelope generator is fed into the operator unit.

More than this, there is an order to the way the SSG-EG update steps during this output cycle interact with the normal envelope generator update cycle. For one of every three output samples, the envelope generator update cycle will occur along with the envelope generator update cycle. When that happens, the operations are performed in the following order:
1. The SSG-EG output cycle update steps are performed
2. The envelope generator update cycle runs
3. The output from the envelope generator is calculated

So, first the SSG-EG update steps occur, followed by the envelope generator update cycle, followed by the output calculation from the envelope generator, which includes applying SSG-EG output inversion if it is active. Where an envelope generator update cycle doesn't occur on the current output cycle, only steps 1 and 3 occur. This order of operations is a relatively minor point, but it does have a measurable effect. In the case where the attenuation level reaches 0x200 in SSG-EG mode, a single output sample is generated at this attenuation level before the envelope loops back to the attack phase.

2. How SSG-EG is updated
I've already explained the basic theory and principles of SSG-EG, and they pretty well cover what SSG-EG mode does, but what they don't cover is exactly how the YM2612 implements them. I think the only way I can adequately explain the implementation of SSG-EG is by providing some sourcecode. This is taken directly from my core, so it's going to have some extra fluff which is specific to my emulator, but it's commented to the point of excess, so it should be easy enough to follow. I've taken great care to try and explain why each step of the SSG-EG update process is implemented the way it is, and what test cases on the hardware validate each step.

This is an example of the steps that need to be carried out for SSG-EG on each envelope generator output cycle, IE, each time the output from the envelope generator is used by the operator unit:

CODE: SELECT ALL

```
//If SSG-EG is enabled, and the current internal attenuation level of the
//envelope generator is greater than or equal to 0x200, we need to run some
//special SSG-EG update steps. Note that hardware tests have proven that
//these operations occur each time the output of the envelope generator is
//calculated (IE, each sample), not just each time the envelope generator
//is updated. The output inversion and phase counter reset steps in particular
//are easily measurable, and have been shown to occur each sample. Note that
//hardware tests have also shown that this update process occurs before the
//normal envelope generator update steps in the case where both run on the
//same cycle. This can allow a single sample to be output at an attenuation
//level of 0x200 before these update steps are applied.
OperatorData* state = &operatorData[channelNo][operatorNo];
if(GetSSGEnabled(channelNo, operatorNo, accessTarget)   //  SSG-EG mode is enabled
```

And this is an example of the steps you need to carry out when actually calculating the output from the envelope generator:

CODE: SELECT ALL

```
//If SSG-EG is enabled and the output is inverted, invert the output data. Note that
//extensive testing has been performed on the hardware to build this implementation.
//This test is performed exactly as shown each time the attenuation value is used.
//Note the way the attack bit is combined with the inversion state. This is known
//to be correct, and is essential in order to deal with cases where the   SSG-EG state
//is changed after key-on. A change in the state of the attack bit will result in an
//immediate inversion of the output. Also note the calculation performed to derive
//the "inverted" data. This calculation has been proven to be binary-accurate.
if(GetSSGEnabled(channelNo, operatorNo, accessTarget)
   && (state->phase != OperatorData::ADSR_RELEASE)
   && (state->ssgOutputInverted ^ GetSSGAttack(channelNo, operatorNo, accessTarget)))
{
   attenuation = 0x200 - attenuation;
```

Note that this code snippet is being performed on a copy of the attenuation value, not the internal attenuation value

itself. IE, this code isn't actually modifying the real attenuation value, just the value which is being output. This inversion of the output also occurs before TL is added to the output attenuation, and before amplitude modulation is applied.

This is an example of the update calculation for the attenuation value in the decay, sustain, and release phase:

CODE: SELECT ALL

```
//Advance the linear decay for the decay, sustain, or release phase. Note
//that if SSG-EG is enabled for this operator, the decay phase runs at 4x
//the normal speed.
if(GetSSGEnabled(channelNo, operatorNo, accessTarget))
{
   //If the current internal attenuation value is below 0x200, advance the
   //decay phase. In most cases when the attenuation level reaches 0x200,
   //the envelope will move back to the attack phase, or the attenuation
   //level will be forced to 0x3FF anyway. One case where neither of these
   //changes occur, and we need this check here to ensure the attenuation
   //level isn't permitted to advance any further, is when we enter an
   //inverted hold phase. Hardware tests have conclusively shown that in
   //these cases, if the internal attenuation level is greater than or equal
```

This is what you need to do when a key-off event occurs:

CODE: SELECT ALL

```
//If SSG-EG is enabled and the output is currently inverted, convert the
//current attenuation value into an equivalent non-inverted value. This
//emulates the release mode behaviour when  SSG-EG is active. Note that we do
//not alter the output inversion flag here. Output inversion is ignored
//during the release phase. The output inversion flag is cleared when key-on
//occurs.
if(GetSSGEnabled(channelNo, operatorNo, accessTarget) && (state->ssgOutputInverted ^
GetSSGAttack(channelNo, operatorNo, accessTarget)))
{
   state->attenuation = Data(attenuationBitCount, 0x200) - state->attenuation;
}
```

And finally, this little command needs to be run each time a key-on event occurs:

CODE: SELECT ALL

```
//Reset the SSG-EG output inversion flag
state->ssgOutputInverted = false;
```

That's my entire SSG-EG implementation. Note that existing emulators, including MAME and Kega, get a lot of the above implementation wrong. In fact, I'm pretty sure every aspect of the SSG-EG implementation in MAME is incorrect, including the calculation to invert the output, so the SSG-EG implementation in MAME really does need to be redone from scratch.

3. SSG-EG test rom
With all this extra information about SSG-EG mode, I think I should post another SSG-EG test ROM to check the behaviour of SSG-EG under some of these undefined circumstances. Here's a test ROM which runs through each of the SSG-EG patterns with an attack phase:
http://nemesis.hacking-cult.org/MegaDri ... elease.bin

Here's the source:
http://nemesis.hacking-cult.org/MegaDri ... elease.asm

Here's the output you get on a PAL MD1600 (WARNING, huge file):
http://nemesis.hacking-cult.org/MegaDri ... elease.rar

And here's the output I currently get from my emulator using the SSG-EG implementation I detailed above:
http://nemesis.hacking-cult.org/MegaDri ... elease.rar

Note that patterns 0x0A and 0x0E alter each time they run. The inversion state may toggle between repetitions of the wave. In the "noise spike" between each repetition, the inversion state is actually being toggled each sample, since the internal attenuation level during the slow attack phase remains above 0x200 for an extended period of time, and the output inversion state is toggled each sample where the alternate bit is set, and the hold bit is unset. The exact sample on which the attenuation level drops below 0x200 determines what the inversion state is set to for the remainder of that repetition.

Interrupt line corrections:
I've got one correction to make regarding the info I posted ages back about the operation of the INT line on the YM2612. I had previously stated this:

"Although not mentioned in the documentation, the reset flag directly affects interrupt generation. When a timer overflows, the YM2612 checks the current state of the timer overflow bit for that timer in the status register. The YM2612 will only generate an interrupt for that timer if the overflow bit is currently unset. If the overflow bit is unset, and the enable flag is set in the timer control register, the INT line will be asserted. The INT line will remain asserted, until the reset bit for that timer is set to 1 in the timer control register, or until the timer overflows again. If the reset bit for the timer is cleared at any time, the interrupt line will immediately be negated. If the timer overflows while INT is still asserted, that is, without reset being written to, the interrupt line will immediately be negated. From this point on, no interrupts will be generated for that timer until the reset bit is set. Once the reset bit has been set when an interrupt has been missed, the INT line will not immediately be asserted. The INT line will remain negated until the timer overflows again."

Everything about the interrupt line not being asserted while the overflow bit is set, or the INT line being negated if the overflow bit hasn't been cleared, is incorrect. There was an error in my testing procedures which caused bad readings. The INT line is asserted whenever a timer overflows, and will remain asserted until the reset bit is flagged for the timer. This makes more sense, and is consistent with how other interrupt devices operate. The INT line is generated in real-time by a logical OR operation between the overflow bits of the two timers.

CSM corrections:
The information I gave previously about CSM mode is basically correct, but there are some points which require clarification. I didn't feel I did enough testing on the interaction between CSM key on/off and manual key on/off events, so I went back and did some more testing. These tests revealed some important details about how CSM mode is implemented in the YM2612.

1. When CSM key-on and key-off occur
I stated previously that CSM key-on and key-off events occurred "at the same time". Obviously this isn't exactly correct, since the key-on event needs to be processed first. Originally, I believed that CSM key-on and key-off occurred during a single envelope generator update cycle, however while implementing CSM support in my core, I came to doubt the accuracy of this. I did some more testing in an attempt to verify whether CSM key-on and key-off occurred on the same update cycle, or whether CSM key-on occurred on one cycle with key-off occurring at the start of the next cycle. What I found is that not only does CSM key-off occur on a separate cycle to key-on, it is possible for a CSM key-on event to be lengthened to any number of update cycles, allowing the note to proceed through the attack, decay, and sustain phases, with CSM key-off not occurring until a register change is made.

The way CSM key on/off events are triggered is simple. First of all, there is a dedicated internal flag which indicates the current "CSM key-on" state. When timer A overflows, if CSM mode is enabled, this "CSM key-on" flag is set. This flag remains set until the next phase generator update cycle, when key-on events are again evaluated. Note that this does occur based on the phase generator update cycle, not the envelope generator update cycle, as elaborated on above regarding when key on/off events are processed. After the key-on state of the operator has been evaluated during the next update cycle, the CSM key-on flag is cleared. If timer A does not overflow again before the next update, the CSM key-on flag will appear negated, which would key-off the operator on the next update cycle after it had been keyed

on by CSM mode. If timer A overflows before every update cycle however, the CSM key-on flag appears to be permanently set, since it is always being re-asserted before it is sampled again. The effect of this is that CSM key-off does not occur, and the key-on event generated by CSM mode allows the operator to proceed through the normal ADSR envelope. Since timer A runs at the same rate as the phase generator update cycle, setting the timer A period to the shortest possible value (0x3FF) will cause this behaviour.

Note that if the channel 3 mode doesn't equal 10 (upper bits of register 0x27 set to 10), the CSM key-on is masked, and the CSM key-on state is forced to 0. This causes a CSM key-off event to occur if CSM mode is disabled while CSM key-on is currently being held on. Note that as I stated in my previous description of CSM mode, CSM mode is only active when the channel 3 mode status bits are set to 10, not when they are set to 11.

2. How CSM key-on is tracked vs manual key-on
There are two separate key-on flags. There is a normal key-on flag, which is updated by sending manual key on/off writes to register $28. There is also a separate CSM key-on flag, which is constantly and automatically updated when CSM mode is enabled, even when an operator has been manually keyed on.

The way these two separate key-on flags interact is really simple. They are OR'd together. An operator is in a state of key-on if either of these flags is asserted. An operator is in a state of key-off if both these flags are cleared. With this implementation, if an operator is manually keyed on, the CSM key-on event is effectively "masked". Since the lines are OR'd together, if the manual key-on line is asserted, it will hold the operator in a keyed on state, regardless of the state of the CSM key-on flag. Once the manual key-on event is cleared, the CSM key-on state is able to have an effect, and change the effective key-on state of the operator. The YM2612 itself only considers that a key on/off event has occurred if the combined OR state of these two flags is different at the beginning of an update cycle than it was at the beginning of the last update cycle.

You should be aware that some unusual combinations of manual key on/off events and CSM mode do require the above details to be implemented correctly. It is possible, for example, to "hand over" a key-on event between CSM mode and manual key on/off events. Eg, you can key on an operator manually, trigger a CSM key-on event in a "held" state (timer A at max rate), key-off the operator manually, and find that the operator does not actually key-off, since the CSM key-on state keeps the operator keyed on. The same rule applies in reverse.

And I think that's all the corrections I'm aware of for the info I've previously posted. The next info I post will be on the LFO. The current LFO implementation in MAME is mostly correct, but there is an error in the phase modulation implementation which I'll detail. I've also figured out the cause of the different sound in Battletech track 12, which was discussed awhile back in this thread. I'll post a detailed description on what causes the difference in sound, but it's actually a fairly complex explanation, so it'll be quite a long post. Apart from that, I've got some more testing to perform on the operator unit relating to the way algorithms are implemented, the DAC needs some attention, and I'm going to publish my own notes on the YM2612 test register, but after that, I'll be pretty much done with the YM2612. Then I can turn my attention entirely to the VDP.

User avatar **TmEE co.(TM)**
Very interested

Sun Feb 22, 2009 4:03 pm

Awesome work

And some relevant stuff, I messed with DAC a little...

Seems that YM2612 misses DAC writes or does not output them when the sample rate is too high... it is odd that when you wait for the chip to be ready, things will not sound too good....
32KHz sounds bad (and its around the rate you get when you wait for the chip). 26KHz (~half the YM2612 sample rate) sounds great though

http://www.fileden.com/files/2008/4/21/ ... CTESTS.RAR