SN76489

Contents

Introduction

This page describes the Programmable Sound Generators based on the SN76489 family of devices. They are almost all identical but produce very different output.

SN76489 sightings

The SN76489AN discrete chip is used in Sega's SG-1000 Mark I and II, and SC-3000 machines, and the Othello Multivision clone. I do not know if the Mark III has a discrete chip or not. The Sega Master System and Game Gear have it integrated into their VDP chips, for backward compatibility to varying extents.

The Mega Drive integrates it into its VDP, although it is often then referred to as an SN76496. It was included to allow for the system's Master System backwards compatibility mode, but was also commonly used because it provides sounds that are impossible to create using the system's main FM synthesis sound hardware (YM2612).

It is used on many of Acorn's BBC and "Business Computer" computers such as the BBC Micro.

The Colecovision uses it too, as a discrete chip as the Colecovision has virtually no custom chips. The Coleco Adam presumably has one too.

Furthermore, it is present in the Sord M5, sold by Takara in Japan and by several others in Europe. Several Memotech home micro computers, such as the MTX-512, included the chip. The SN76489 was also used in a modified form, with the designation changed to TMS9919 to fit in the 99xx series (from which the SC-3000 VDP comes as well), in the TI-99/4A.

Other computers thought to use the chip are:

- Hanimex Pencil
- Video Technology CreatiVision

It is undoubtedly used in a LOT of arcade machines. This is a partial list; numbers in brackets signify how many SN76489s are present:

- Bank Panic (3)
- Champion Boxing
- Champion Pro Wrestling
- Gigas (4)
- Gigas Mark II (4)
- Free Kick
- Lady Bug (2)
- Mr. Do! (2)
- Mr. Do's Castle (4)
- Mr. Do's Wild Ride (4)
- Super Locomotive (2)
- Wonder Boy: Monster Land (2)

A clone of the SN76489 is included in the Tandy 1000 home computer, for compatibility with the one in the IBM PCjr of which it is itself a clone.

Accessing the SN76489 from software

The SN76489 has an 8-bit write-only data bus, so it is controlled in software by writing bytes to it. How this is done depends on the system.

- Sega Game 1000 (SG-1000)
- Sega Computer 3000 (SC-3000)
- Sega Master System (SMS)
- Sega Game Gear (GG)
- Sega Mega Drive/Genesis (in Master System compatibility mode)
  The SN76489 can be accessed by writing to any I/O port between 0x40 and 0x7f, although officially only 0x7f was recommended. A few games write to 0x7e.

- Sega Mega Drive/Genesis
  The SN76489 is memory-mapped to the 68000 CPU at location 0xc00011 (mirrored at 0xc00013, 0xc00015, 0xc00017) and the Z80 CPU at 0x7f11 (mirrored at 0x7f13, 0x7f15, 0x7f17).

- ColecoVision
- Coleco Adam?
  The SN76489 is mapped to I/O port $ff.

- Tandy 1000
- IBM PCjr?
  The NCR 8496 is mapped to I/O port(?) $c0.

- Other systems
  Let me know :)

SN76489 registers

The SN76489 has 8 "registers" - 4 x 4 bit volume registers, 3 x 10 bit tone registers and 1 x 3 bit noise register. Of course, for hardware reasons these may internally be wider.

Channel	*Volume registers*	*Tone & noise registers*
0	Vol0	Tone0
1	Vol1	Tone1
2	Vol2	Tone2
3	Vol3	Noise

Volume registers
  The value represents the attenuation of the output. Hence, %0000 is full volume and %1111 is silence.

Tone registers
  These give a counter reset value for the tone generators. Hence, low values give high frequencies and vice versa.

Noise register
  One bit selects the mode ("periodic" or "white") and the other two select a shift rate.

One bit selects the mode ("periodic" or "white") and the other two select a shift rate.

It appears the initial state of these registers depends on the hardware:

- Discrete chips seem to start with random values (an SC-3000 is reported to start with a tone before the chip is written to by the software).
- The Sega integrated versions seem to start initialised with zeroes in the tone/noise registers and ones in the volume registers (silence).

SN76489 register writes

When a byte is written to the SN76489, it processes it as follows:

If bit 7 is 1 then the byte is a LATCH/DATA byte.

```
%1cctdddd
  |||````-- Data
  ||`------ Type
  ``------- Channel
```

Bits 6 and 5 (`cc`) give the channel to be latched, ALWAYS. This selects the row in the above table - %00 is channel 0, %01 is channel 1, %10 is channel 2, %11 is channel 3 as you might expect.
Bit 4 (`t`) determines whether to latch volume (1) or tone/noise (0) data - this gives the column.

The remaining 4 bits (`dddd`) are placed into the low 4 bits of the relevant register. For the three-bit noise register, the highest bit is discarded.

The latched register is NEVER cleared by a data byte.

If bit 7 is 0 then the byte is a DATA byte.

```
%0-DDDDDD
  |``````-- Data
  `-------- Unused
```

If the currently latched register is a tone register then the low 6 bits of the byte (`DDDDDD`) are placed into the high 6 bits of the latched register. If the latched register is less than 6 bits wide (ie. not one of the tone registers), instead the low bits are placed into the corresponding bits of the register, and any extra high bits are discarded.

The data have the following meanings (described more fully later):

Tone registers

```
DDDDDDdddd = cccccccccc
```

- `DDDDDDdddd` gives the 10-bit half-wave counter reset value.

Volume registers

```
(DDDDDD)dddd = (--vvvv)vvvv
```

- `dddd` gives the 4-bit volume value.
- If a data byte is written, the low 4 bits of `DDDDDD` update the 4-bit volume value. However, this is unnecessary.

Noise register

```
    (DDDDDD)dddd = (---trr)-trr
```

- The low 2 bits of `dddd` select the shift rate and the next highest bit (bit 2) selects the mode (white (1) or "periodic" (0)).
- If a data byte is written, its low 3 bits update the shift rate and mode in the same way.

This means that the following data will have the following effect (spacing added for clarity, hopefully):

```
%1 00 0 1110      Latch, channel 0, tone, data %1110
%0 0  001111      Data %001111
```

Set channel 0 tone to %0011111110 = 0xfe (440Hz @ 3579545Hz clock)

```
%1 01 1 1111      Latch, channel 1, volume, data %1111
```

Set channel 1 volume to %1111 = 0xf (silent)

```
%1 10 1 1111      Latch, channel 2, volume, data %1111
%0 0  000000      Data %000000
```

Set channel 2 volume to %1111 = 0xf (silent) THEN update it to %0000 = 0x0 (full) ***The data byte is NOT ignored.*** If it is, you will hear a sustained tone while reading a message box in Alex Kidd in Miracle World.

```
%1 11 0 0101      Latch, channel 3, noise, data %0101
```

Set noise register to %101 (white noise, medium shift rate)

```
%1 11 0 0101      Latch, channel 3, noise, data %0101
%0 0  000100      Data %000100
```

Set noise register to %101 (white noise, medium shift rate) THEN update it to %100 (white noise, high shift rate) ***The data byte is NOT ignored.*** If it is, some games (e.g. Micro Machines) produce the wrong sound on their noise channel.

Also of note is that the tone registers update immediately when a byte is written; they do not wait until all 10 bits are written.

Data written	*Tone0 contents*
`1 00 0 0000`	`------0000`
`0 0 000000`	`0000000000`
`1 00 0 1111`	`0000001111`
`0 0 111111`	`1111111111`

- signifies an unknown bit (whatever was previously in the register)

There were a couple of ways to handle SN76489 writes in older, inaccurate emulators:

1. Latch only the tone registers, as above, and leave them latched when other types of data (volume, noise) are written. This gives a "squawk" effect on SMS Micro Machines' title screen, which drowns out the "eek".

2. Latch tone registers as above, and "unlatch" when other types of data are written. When a data byte is written with it unlatched, the data is discarded. This fixes the "squawk" but leaves the "eek".

How the SN76489 makes sound

This is already well documented, but I'll repeat it again with (hopefully) a more hardware-related perspective.

The SN76489 is connected to a clock signal, which is commonly 3579545Hz for NTSC systems and 3546893Hz for PAL/SECAM systems (these are based on the associated TV colour subcarrier frequencies, and are common master clock speeds for many systems). It divides this clock by 16 to get its internal clock. The datasheets specify a maximum of 4MHz.

Some versions (specified as the SN76489N in the datasheets) instead have a divider of 2 and a maximum clock of 500kHz, giving an equivalent post-divide clock rate.

For each channel (all 4) there is a 10 bit counter, and an output bit. Each clock cycle, the counter is decremented (if it is non-zero). If, after being decremented, it is zero, the following happens:

Tone channels

The counter is reset to the value currently in the corresponding register (eg. Tone0 for channel 0). The output bit is flipped - if it is currently outputting 1, it changes to 0, and vice versa. This output is passed to the mixer (see below). The initial output value may be arbitrarily set.

So, it produces a square wave output with wavelength twice the value in the corresponding register (measured in clock ticks). The frequency of this can be calculated by

```
                     Input clock (Hz) (3579545)
   Frequency (Hz) = ----------------------------------
                  2 x register value x divider (16)
```

Example values for an NTSC-clocked chip are given and are generally assumed throughout. Thus, for example, 0x0fe gives 440.4Hz.

If the register value is zero or one then the output is a constant value of +1. This is often used for sample playback on the SN76489.

Tone range

The lowest possible tone, using register value $3ff, is 109Hz (assuming an input clock of 3579545Hz), which corresponds to MIDI note A2 -10 cents.

The highest possible tone, using register value $001, is 111861Hz, which corresponds to MIDI note D10 -14 cents. However, in practice, smoothing capacitors and other, perhaps less deliberate, imperfections in the output mean that such a high note is not audible; in tests on an

SMS2, the highest note that gave any audible output was register value $006, giving frequency 18643Hz (MIDI note A12 -12 cents). Thus, there is effectively a range of 10 octaves.

Noise channel

The counter is reset according to the low 2 bits of the noise register as follows:

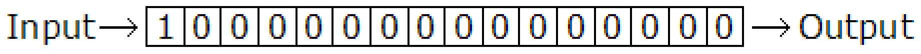
Low 2 bits of register	Value counter is reset to
00	0x10
01	0x20
10	0x40
11	Tone2

As with the tone channels, the output bit is toggled between 0 and 1. However, this is not sent to the mixer, but to a "linear feedback shift register" (LFSR), which can generate noise or act as a divider.

The Linear Feedback Shift Register

The LFSR is an array of either 15 or 16 bits, depending on the chip version; a 16-bit version can give the same output as a 15-bit one with adjustment of parameters.

When its input changes from 0 to 1 (ie. only **once** for every two times the related counter reaches zero), the array is shifted by one bit; the direction doesn't matter, it just changes what numbers you use, so I will arbitrarily say it shifts right. The bit that is shifted off the end (either 0 or 1) is output to the mixer.



The input bit is determined by an XOR feedback network. There are two types: an external network, where the XOR gates are external to the shift register, and internal, where they are between bits. Both are discussed below. Certain bits are used as inputs to the XOR gates; these are the "tapped" bits. An n-bit shift register can generate pseudo-random sequences with periodicity up to $2^n - 1$, depending on the tapped bits.

The external LFSR type is discussed below.

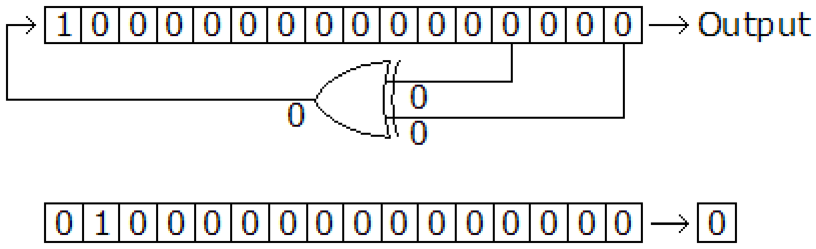
I will add more on the internal LFSR later - Maxim June 04, 2005, at 08:09 AM~

For white noise (Noise register bit 2 = 1):

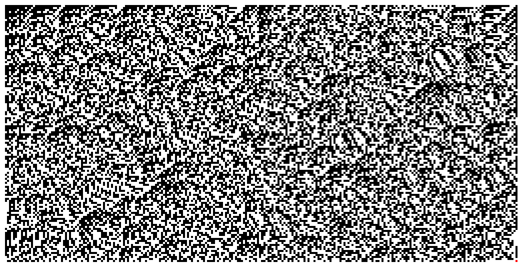
For the SMS (1 and 2), Genesis and Game Gear, the tapped bits are bits 0 and 3 ($0009), fed back into bit 15. For the SG-1000, OMV, SC-3000H, BBC Micro and Colecovision, the tapped bits are bits 0 and 1 ($0003), fed back into bit 14. For the Tandy 1000, the tapped bits are bits 0 and 4 ($0011), fed back into bit 14.

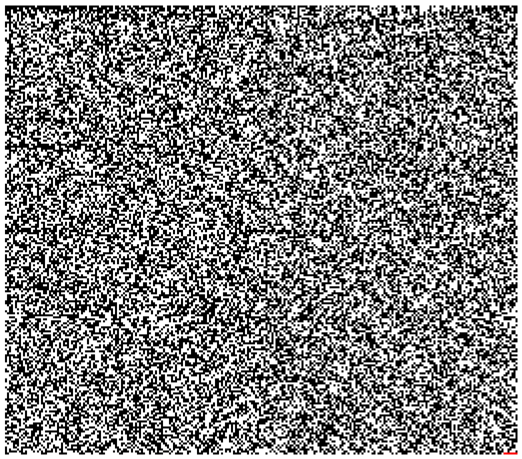
I would like to confirm the bit pattern for other systems, please contact me if you can help by running/coding homebrew code on a real system and sampling the sound. - Maxim

Example (SMS/GG):



The output bit patterns are shown graphically below. (Red pixels are used to pad to rectangular sizes.) Click to download the bitpatterns, padded to 8 bits.


SN76489 white noise

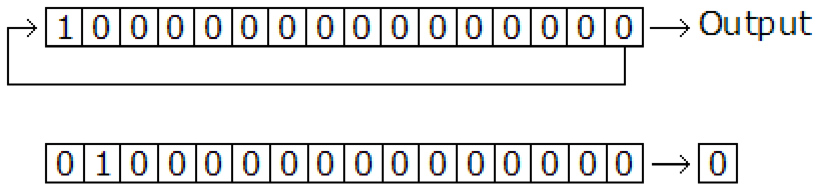

Sega VDP white noise

For "periodic noise" (Noise register bit 2 = 0):

For all variants, only bit 0 is tapped, ie. the output bit is also the input bit. The effect of this is to output the contents of the shift register in loop of the same length (16 bits for the SMS (1 and 2), Genesis and Game Gear, 15 bits for SG-1000, OMV, BBC Micro, SC-3000H, ColecoVision and Tandy 1000; other systems need investigation).

When the noise register is written to, the shift register is reset, such that all bits are zero except for the highest bit. This will make the "periodic noise" output a 1/16th (or 1/15th) duty cycle, and is important as it also affects the sound of white noise.

Thus, the output in "periodic noise" mode will also be at a fraction of the frequency of the underlying driving signal (discussed above). For a 16-bit shift register and a 3759545Hz input clock, this gives "periodic noise" a frequency range of 6.8Hz to 6991Hz (when using tone channel 2 as the driving signal, with register values $3ff and $001 respectively), a range of 10 octaves (MIDI notes A-2 to A8), shifted 4 octaves down from the regular tone range.



Note that this "periodic noise", as it is called in the original chip's documentation, is in fact not periodic noise as it is defined elsewhere (white noise with a configurable periodicity); it is a duty cycle modifier. For this reason, throughout this document it is always referred to with quotes.

Output inversion

Some systems seem to produce inverted output, which can confuse matters when reverse-engineering the noise generator as `1`s become `0`s and vice versa. This is why you may see conflicting information from other sources. It is of note that a 16-bit LFSR with white noise feedback pattern $0006 can be inverted by using feedback pattern $8005 instead.

An implementation of the noise shift register

```
    ShiftRegister=(ShiftRegister>>1) |
                 ((WhiteNoise
                   ?parity(ShiftRegister&TappedBits)
                   :ShiftRegister&1)<<15);
    Output=ShiftRegister&1;
```

Get Code

where `parity()` is a function that returns 1 if its (16-bit unsigned int) parameter has an odd number of bits set and 0 otherwise; and TappedBits depends on the system being emulated (and so should be a variable, not a constant, for any emulation that is supposed to handle more than one of the known feedback types), for example 0x0009 for the Master System (bits 0 and 3 set). Here is a fast SIMD implementaion of 16-bit `parity()`:

```
int parity(int val) {
    val^=val>>8;
    val^=val>>4;
    val^=val>>2;
    val^=val>>1;
    return val&1;
};
```

Get Code

Thanks go to Dave (finaldave) for coming up with this. You may get faster results with expressions tailored to certain common feedback patterns, and of course if you can use assembler to access a CPU's built-in parity checking instructions/flags.

Volume/attenuation

The mixer then multiplies each channel's output by the corresponding volume (or, equivalently, applies the corresponding attenuation), and sums them. The result is output to an amplifier which outputs them at suitable levels for audio.

The SN76489 attenuates the volume by 2dB for each step in the volume register. This is almost completely meaningless to most people, so here's an explanation.

The decibel scale is a logarithmic comparative scale of power. One bel is defined as

```
    power 1
log -------
    power 2
```

Whether it's positive or negative depends on which way around you put power 1 and power 2. The log is to base 10.

However, this tends to give values that are small and fiddly to deal with, so the standard is to quote values as decibels (1 decibel = 10 bels). Thus,

```
                 power 1
decibels = 10 log -------
                 power 2
```

One decibel is just above the threshold at which most people will notice a change in volume.

In most cases we are not dealing with power, we are instead dealing with voltages in the form of the output voltage being used to drive a speaker. You may remember from school that power is proportional to the square of the voltage. Thus, applying a little mathematical knowledge:

```
                 (voltage 1)'^2^'          voltage 1
decibels = 10 log ------------ = 20 log ---------
                 (voltage 2)'^2^'          voltage 2
```

Rearranging,

```
voltage 1     (decibels / 20)
--------- = 10
voltage 2
```

Thus, a drop of 2dB will correspond to a ratio of $10^{-0.1}$ = 0.79432823 between the current and previous output values. This can be used to build an output table, for example:

```
int volume_table[16]={
  32767, 26028, 20675, 16422, 13045, 10362,  8231,  6568,
   5193,  4125,  3277,  2603,  2067,  1642,  1304,     0
};
```

Get Code

These correspond to volume register values 0x0 to 0xf, in that order.

The last value is fixed to zero, regardless of what the previous value was, to allow silence to be output.

Depending on later hardware in the chain between the SN76489 and your ears, there may be some distortion introduced. My tests with an SMS and a TV card found the highest three volume levels to be clipped, for example.

The imperfect SN76489

Real components aren't perfect. The output of the SN76489 in its various implementations can be severely affected by this.

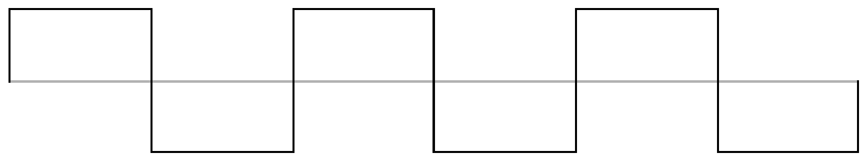
Wherever a voltage (output) is artificially held away from zero, there will be leakage and the actual output will decay towards zero at a rate proportional to the offset from zero:

```
    dV
    -- = -kV
    dt
```

where `k` is a constant

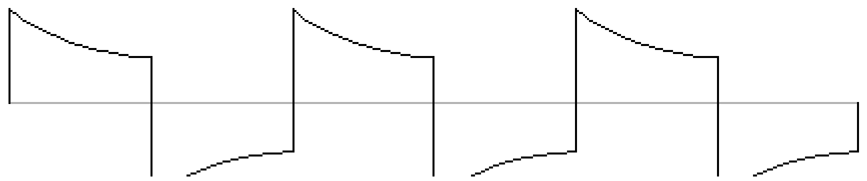
This affects the output from the SN76489 both internally (for the outputs from the wave generators to the mixer) and externally (for the output of the mixer).

The effect on the tone channels is to change the shape of their output waves from this:



Note: this diagram needs to be updated. The input seems to actually be 0/1, but the output is centred around zero.

to something like this:



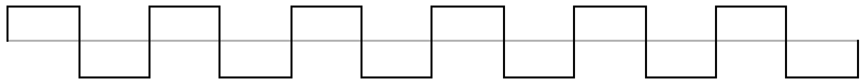
This needs to be replaced with a real output sample, for authenticity and accuracy.

If the tone register value is large enough, they will decay close to zero.

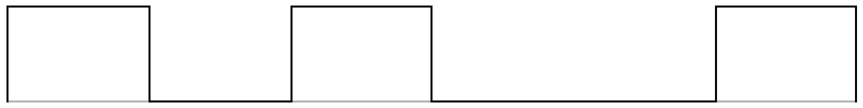
If the tone register value is zero, the constant offset output will just decay to zero. However, whenever the volume of the output is changed, the constant offset is restored. This allows speech effects.

The effect on the noise generator's output is this:

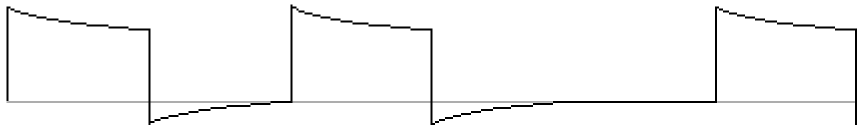
Signal generator output, for reference:



Perfect output (assuming output bit pattern of 101001):



Actual output:



This needs to be replaced with a real output sample, for authenticity and accuracy.

The empirical effects are:
- The sound of tones is changed very slightly
- Noise sounds a bit louder
- Voices sound slightly different

Some games were programmed with the SN76489 considered to be a perfect device, for example for PCM voice effects.

Emulating imperfection

In most cases, emulating the imperfections of the SN76489 is processor-intensive and difficult to get quantitatively correct. However, a "perfect" emulation is likely to be sufficiently far removed from the actual device to give output that is noticeably "wrong".

The tone channels are an excellent example of this. Because of the voltage decay mentioned above, their outputs degrade from a rectangular wave with positions 0 and +1 to one that more closely resembles one with positions -0.5 and +0.5, with some curvature. This is because the average value of the output bit is 0.5, and voltage decay will lead to this average offset from zero disappearing. Thus, emulating the tone channel outputs as -0.5/+0.5 instead of 0/+1 gives a much more pleasing representation.

The noise channel can be emulated the same way, or it can be left as 0/+1; because of its randomness, the effect is less. However, for "periodic" noise, where the average value of the output bit is 1/16 or 1/15, the latter case is a better approximation.

Playing samples on the PSG

This is for the reference of those wishing to put sample playback in their demos, and for those whose sound core doesn't do voices. Emulator authors may wish to add implementation suggestions.

Sample playback is possible on the SN76489 but not the YM2413 FM chip.

Sample playback makes use of a feature of the SN76489's tone generators: when the half-wavelength (tone value) is set to 1, they output a DC offset value corresponding to the volume level (i.e. the wave does not flip-flop). By rapidly manipulating the volume, a crude form of PCM is obtained.

(Note that this may be a feature of Sega's implementation of the SN76489. It does not appear in any Sega 8-bit code designed for older systems with the standard chip so it may need to be confirmed by experiment.)

Because the volume levels are non-linear, and only have four bits of resolution, it makes the quality of reproduction rather poor. Furthermore, there is no facility to stream data to the SN76489 so it must be done by the CPU; in almost all games this means the game is frozen while samples are played. In the few games that do some work during sample playback, the sample playback quality is usually made even worse.

Sega 8-bit games play their sampled audio in a few different ways:

Simple 4-bit PCM

A stream of 4-bit linear PCM data is read from ROM (packed two samples to a byte), and emitted as SN76489 attenuation commands to one or more of the tone channels. This results in a non-linear output which can make samples sound quieter than expected.

This can be ameliorated by pre-processing the data to account for the non-linear response; however, very few games do this.

1-bit PCM

A stream of 1-bit PCM data is read from ROM (packed eight samples to a byte), and emitted as either no, or maximum, SN76489 attenuation commands to one or more of the tone channels. This results in a faithful representation of the data, but the dynamic range of 1-bit audio is extremely poor so the result is not very good. Typically the samples seem to have been heavily amplified and clipped, resulting in loud samples.

Lookup for 8-bit PCM

Lookup for 8-bit PCM

A stream of 8-bit PCM data is read from ROM (one sample per byte) and used to look up a triplet of SN76489 attenuation commands from a table in ROM. These are emitted in close succession. By careful construction of the lookup table, the commands are able to address a large number of volume levels by combining the non-linear volumes.

During the transition from one sample to another, this can produce unwanted artefacts because the intermediate total attenuation may not lie between the start and end points. For example, transitioning from attenuations 4,0,0 (total output level 79.9%) to 2,1,0 (total output level 80.1%) may temporarily be in the state 2,0,0 (total output level 87.7%). This can be avoided by minimising the transition time, but seems to still produce noise.

Methods not found in games

- The 1-bit approach could be used for Pulse-density modulation, which promises much higher clarity. At very high sampling rates it may exceed the quality of PCM, thanks to noise shaping, but at the highest rates achievable on the Master System/Game Gear the result is quite noisy.

- Pulse Width Modulation is a special case of Pulse Density Modulation and is not very useful.

- Preprocessing of the data to have a low-frequency DC offset can allow moving the range of the high-frequency waveform into the area of the non-linear response where the precision is greater. This can particularly help with samples with a large dynamic range. See this post by blargg for examples and a program to do the conversion.

- The issues with noise in the 8-bit lookup method could be ameliorated by instead storing a 12-bit stream of attenuation command triplets, each optimised to avoid out-of-range transitions from the previous sample.

Sample preprocessing

The results of sample playback can be improved by better preprocessing of the data. It should at least be normalised to 100% to allow the best use of the output range. The dynamic range can be compressed to make the sample sound louder, and make quiet sounds more reproducible. Quiet parts should be silenced or offset to avoid a 1-bit noise floor in a 4-bit sample.

Uniformity of playback

In almost all cases, the underlying audio data will be based on a uniform sampling rate. If it is played back at a non-uniform rate, due to branches in the code (for example to retrieve the next byte for <8bit samples), it may produce unwanted effects. In practice, however, this seems not matter very much and many games have some small non-uniformity. It is usually possible to cancel out the branching effect by adding some time-wasting opcodes to the faster branches.

Sampling rates

For all sample playback methods, there is generally an improvement in quality as the sampling rate goes up; but this has a high cost for ROM space. No game plays samples as fast as the CPU can go, they all include some sort of busy wait to limit the rate.

The lowest quality audio seen in games is around 4kHz at 1 bit, which can fit 32.8s of audio in 16KB. The highest is around 21kHz at 4 bits, which can fit 1.6s of audio in 16KB.

Game Gear stereo extension

When a byte is written to port 0x06 on the Game Gear, the PSG output is affected as follows:

Bit	*Channel*	*Side*
0	0	Right
1	1	Right
2	2	Right
3	3	Right
4	0	Left
5	1	Left
6	2	Left
7	3	Left

If a bit is set, the corresponding channel is output to the corresponding side. So, `0xff` outputs all channels to all sides, `0xf0` outputs to the left side only, etc.

History

6/6/2002
: Clarification that SN76489 tones update immediately after latch byte. Use of 2-stage volume writes found.

22/8/2002
: Charles MacDonald sampled GG and Genesis noise for me, it's the same bit pattern as SMS noise.

20/10/2002
: Fixed some typos.

21/3/2003
: Added SC-3000H noise feedback pattern, thanks to Charles MacDonald for getting the data for me.

21/4/2003
: Charles MacDonald sampled SMS1 noise, it's the same bit pattern as the SMS2, GG and Genesis.

27/4/2005
: Added sections on 15-bit shift registers and volume/attenuation. Most sections tweaked, clarified, corrected and extended.

1/5/2005
: Added to Wiki.

21/5/2005
: Added some sightings.

23/5/2005
: Added ColecoVision noise information (same as BBC Micro), thanks to Daniel Bienvenu.

12/11/2005
: Added OMV information, and clarified 15/16 bit shift register differences.

Credits

Based on research by Maxim, after initial results by John Kortink. BBC Micro noise data thanks to John Kortink. SMS1, Game Gear, Genesis and SC-3000H noise data thanks to Charles MacDonald. ColecoVision noise data thanks to Daniel Bienvenu.

- Back to Sound index
- Back to Development index

3

Return to top

0.538s