Notes on
Ensoniq DOC*

March 5, 1986

Writer: Allen Watson
Apple Technical Publications

*Digital Oscillator Chip

Background

Who: Rob Moore, Gus Andrade, Allen Watson

What: Visit to the Ensoniq factory in Malvern, Pa.

When: February 25th and 26th, 1986

Why: To learn more about the IC (DOC) used in the Ensoniq Mirage

How: By spending two days in intensive discussions with the DOC's designers

Ensoniq People

Al Charpentier

Bill Mauchly

Alex Limberis, Mgr., Software Development

Bob Yannes, Sr. Design Engineer

Bruce Crockett

Tom Metcalf, Sound Designer

Hardware

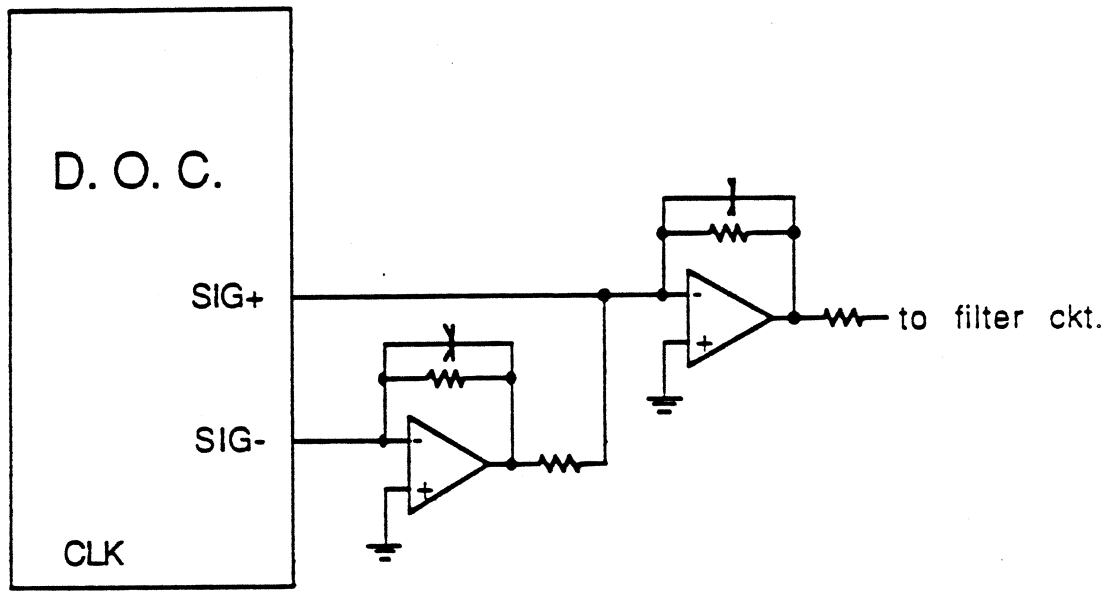
After arriving at Ensoniq and being indtroduced, we began discussing hardware issues. Rob sketched a schematic diagram showing the way we at Apple are using the Ensoniq IC. Our clock frequency is 7.14 MHz, somewhat slower than the nominal value, 8 MHz. The slower clock reduces out sample rate (with all 32 oscillators active) to about 26.3 kHz.

The slow clock also lengthens the refresh cycle for the dynamic RAM used for the wave tables. Bob Yannes assured us that would not cause any problems, even though the resulting refresh cycle is slightly longer than the maximum specified for the RAM chips (4ms at 70°C).

Bob told us that most of what we were doing was OK, but he pointed out a couple of places where we could make changes that would improve performance.

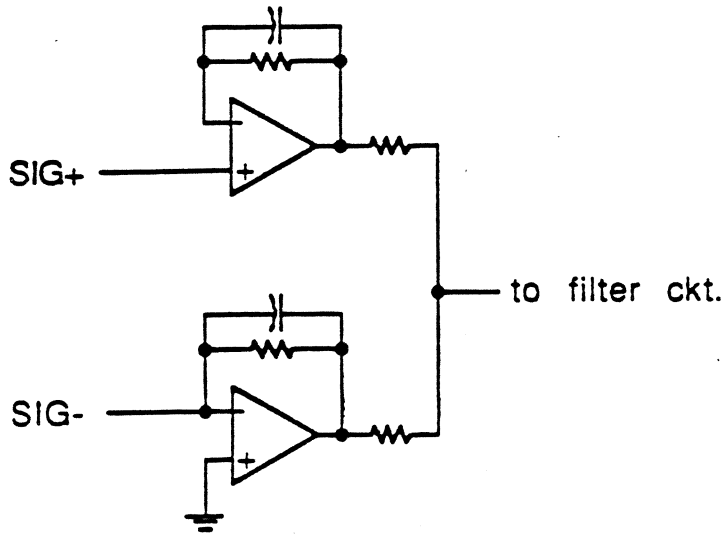
One of the changes Bob suggested was in the way we combine the differential outputs from the chip. Rob explained that we had started out by trying to use just one, but shifts in the DC offset of the signal at the start and end of notes cause unwanted clicks in the sound. We added an inverter to the SIG– output and combined the inverted signal with the direct SIG+ output. The trouble with that hookup is the amplifier time delay in the SIG– signal, compared with the direct SIG+ signal. The time differences between the two signals produce transient distortion in the sum. (See Figure 1.)

Figure 1. Present Circuit for Summing Outputs



Bob suggested that we modify the circuit so that both differential outputs go through amplifiers before the outputs are summed. (See Figure 2.) Rob explained that our board design was essentially frozen, but that in his opinion we could make such a change.

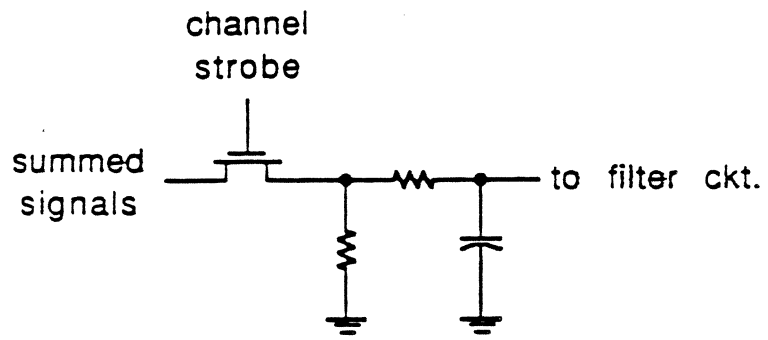
Figure 2. Recommended Circuit for Summing Outputs



Rob described our low-pass filter (four poles, with cutoff at about 10 kHz) and asked about the use of a sample-and-hold circuit ahead of the filter. He said that we had tried such a circuit but that it didn't work very well. Apparently the differential time delays shorten the time during which both signals are valid. Also, the last oscillator gets held for a longer time than the others, because of the chip's refresh cycle, making its contribution to the overall sound level disproportionately large.

Bob explained that Ensoniq didn't mean to recommend the use of a true sample and hold, but merely a gate. [N.B. This is an example of the way careless terminology causes problems.] The gate circuit shown in Figure 3 passes a short pulse of valid signal each time the channel strobe turns on the FET. Assuming the RC time constant of the gate is short enough that each pulse decays all the way to zero before the next sample, each oscillator contributes its rightful share to the overall signal level.

Figure 3. Gate Circuit



Bob also pointed out that the gating circuit isn't necessary except when running the chip in envelope mode (which we probably won't do), so we don't need it.

Rob then asked about resetting the chip. Bob told us the chip uses dynamic registers, and that resetting turns them all off. When off, they don't get refreshed, so the effect is the same as setting them to all ones.

Those registers full of ones have been causing problems for us, because they enable interrupts from the chip. Before the chip will operate properly, we have to select the number of oscillators we'll be using (so that the chip will refresh them). To eliminate the spurious interrupts, we have to read the chip and clear the interrupt once for each of the 32 registers.

The DOC's Processor

The Digital Oscillator Chip (DOC) contains 34 words of 80 bits each, one per oscillator plus two more for refresh operation. The chip's ALU operates on one 80-bit word at a time, doing everything in one processing cycle, so it takes 34 cycles to update all of the oscillators (assuming all of them have been selected). With the normal 8 MHz clock, one processing cycle lasts one microsecond, so an update cycle takes 34 microseconds, making the sample rate for each oscillator equal to about 29.4 kHz.

Question: should we compress the dynamics when we record our sound samples. Rob suggested a two-track sampling technique: separate the waveform and the envelope, then use compression on the waveform. Ensoniq does something like this, then synthesizes the envelope. They repeat the same waveform during the decay portion of the sound.

Our tools can take care of some of the resolution and scaling, and the addresses where the samples are stored, along with table sizes and such. We'll need an added level of control interface, one that's more appropriate for developers.

Bob Yannes pointed out that their synth-type machine uses the chip's sync and modulation modes to produce some very complex waveforms.

DOC Operating Modes

The DOC has four modes of operation. They are listed in Table 1.

Table 1. Operating Modes

Mode Bits	Mode Selected	Explanation
00	Free-Run Mode	Continuous repetition of same wave data
01	One-Shot Mode	One time through the wave data
10	Sync/AM Mode	
11	Swap Mode	Two oscillators cross-coupled ("osc-couples")

The size of the wave table, in pages, is normally a power of two. For samples of other lengths, the wave data must be followed by zeroes. Because of skip counting, it takes eight zeroes in a row to guarantee that the oscillator will stop.

Ending a wave: There is a difference between address wrap-around and halt. An oscillator can transfer control to a second one only when it halts; thus, swap mode requires that at least the first oscillator be in one-shot mode.

Swap Mode

The most-used mode is swap mode, which uses a pair of oscillators. One way of using swap mode is to break a complex sound, such as a piano note, into two sections. The first oscillator generates the initial portion of the sound as a single one-shot sequence. The first oscillator is in one-shot mode. When it finishes, it automatically starts a second oscillator that runs in free-run (loop) mode, repeating a single wave (or other short segment) while the sound decays.

Another form of swap mode is to have a pair of oscillators swap back and forth indefinitely. Both oscillators must be in one-shot mode so that each one will stop; an oscillator in free-run mode won't stop, so it won't start the other oscillator.

Sync/AM Mode

This is two modes in one, but determining which one you get is somewhat bizarre. As in swap mode, you use an ajacent-numbered pair of oscillators. If the lower number is odd, the first (odd-numbered) oscillator controls the timing of the second (even-numbered) oscillator. If the lower number is even, the first (even-numbered) oscillator controls the amplitude of the second (odd-numbered) one. Thus, to select sync mode, you have to use an odd/even pair, and to select AM mode, you use an even/odd pair.

Sync Mode

In sync mode, the output comes from the second (even-numbered) oscillator, but the timing is controlled by the first (odd-numbered) one. When the first oscillator wraps, it resets the second one to the beginning of its wave table. (If the first oscillator enounters zeroes in its wave table, it will halt.) The first oscillator can interrupt the processor when it wraps, making it possible for the program to change its parameters.

The sounds produced by sync-mode operation can be quite surprising. For example, sweeping the frequency of the first oscillator seems to lock in on successive overtones of the sound generated by the second one.

AM Mode

In AM mode, the output comes from the second (odd-numbered) oscillator, but the envelope is provided by the first (even-numbered) one. In this mode, you can't adjust the volume of the output; the volume-control multiplier is handling the modulation. (This mode is the envelope mode referred to in the earlier discussion about the sample and hold.)

As the name implies, the modulation produced in AM mode is plain, garden-variety amplitude modulation. Ensoniq sometimes refers to it as ring modulation, which is something quite different. Musicians apparently use the term *ring modulation* rather loosely. (Four-quadrant multiplication of two signals produces ring modulation, while two-quadrant modulation produces amplitude modulation. In amplitude modulation, the carrier signal is present in the output; in ring modulation, the carrier is absent. If a ring modulator is not perfectly symmetric, a small amount of carrier will leak through into the output. The first ring modulators were sometimes called *balanced modulators*, semantic evidence of the difficulty of obtaining this symmetry.)

Recommended Reset Sequence

To make sure everything gets set up properly when starting up, follow this sequence.

(1) Set the number of oscillators

(2) Clear interrupts (in the DOC)

(3) Set all oscillators to HALT

(4) Read oscillator IRQ registers

(5) Write zeroes in other oscillator registers

(6) Fill memory with the value $80 (center value, for zero output level)

Note about values: the sampled data ranges from a minimum of $01, through a center (zero) value of $80, to a maximum of $FF. The excursions can range from -127 to +127. Hexadecimal $00 corresponds to a sample value of -128, which never occurs, so we can use $00 as a special case, that is, to mark the end of the data.

Resolution Versus Table Size

The Ensoniq chip has two parameters that pertain to the size of the wave table: they are **resolution** and **table size**. Depending on what you are trying to accomplish, you may or may not want these two parameters to be proportional when changing to different-sized wave tables.

If you want the same note frequency despite changing sizes of tables, you have to step through by different-sized steps. If you change resolution to track changes in table size, the chip adjusts the sample skipping to produce the same frequency. *** **Did I get this backward?** ***

For sampled sounds, different-sized tables represent different frequencies (different-length sounds).

Generally, long-sample sounds are in one-shot mode (and need interrupts). They often won't be exactly a (power-of-two) page long. The sound can have a glitch on when you loop, because you have to reset and restart at zero.

Short-sample sounds are in free-run mode, which uses true wrap-around of addresses and so does not produce a glitch.

(At this point there was a discussion of whether we were going too fast, and specifically whether we needed a refresher course in skip-sampling. Nobody was willing to admit to needing it, but we decided to do it anyway.)

Sidebar: Skip-Sampling Fundamentals

There are two ways of controlling the frequencies of notes generated from a wave-shape stored as sample values (waveform table).

- Play back the samples at different sample frequencies.
- With the sample frequency the same, change the number of samples to skip.

The second method is simpler to implement, but it can create several kinds of errors. For example, if the desired frequency is not a simple ratio of the sample frequency, you need to be able to skip by fractions. You have to use one of the samples you have, so you use the nearest one.

If you had more samples, you'd be able to minimize the error caused by the need for fractional samples. There's a limit to how far you can take this: eventually you reach the limit of resolution of your DAC (or your sample size, if not the same), so your additional samples end up repeating the same values. If your word-size permits it, you can pre-process your wave data and interpolate additional values between the samples you started with.

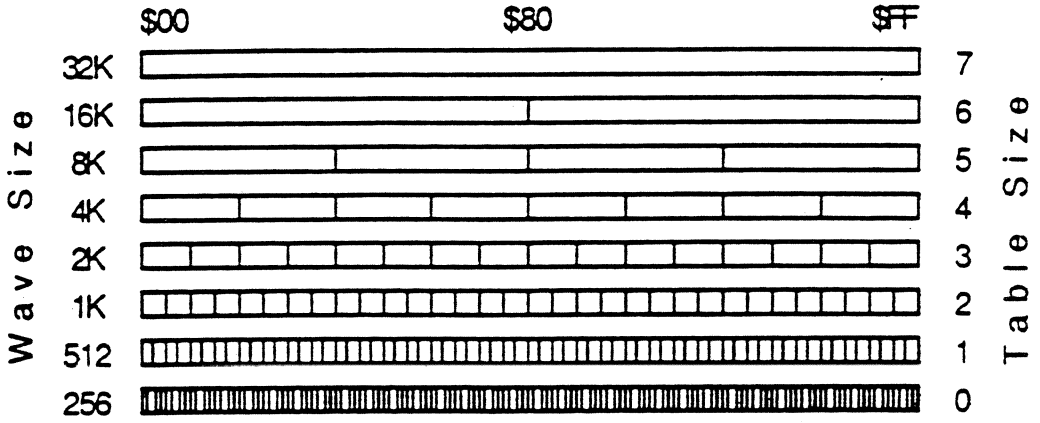
Of course, for skip-sampling to work at all, you have to use a skip (address) counter with high resolution (arithmetical precision) and carry along any fractional part to the next skip.

The Pyramid Scheme

Wave tables have to start on page boundaries, and their sizes, in pages, have to be powers of two. These restrictions make it hard to store many large wave tables without wasting a lot of memory. The Pyramid Sceme is a method of allocating storage so as to use memory as efficiently as possible.

For different sizes of wave tables, we can subdivide 64K bytes of memory in several different ways, as shown in Figure 4.

Figure 4. Pyramid Sceme for Wave-Table Storage



The wave-table address generator is an accumulator. It alwa·· starts at zero, but it can have additional high-order bits as a constant offset. In other rds, the address generator can scan a block whose size is equal to wave size, starting on any even multiple of wave size. For a given starting page, find the largest wave-size that can start on that address.

The general idea here is to break a long wave table into a small number of chunks that the address generator can handle, then generate the sound using swap mode, alternating between the two oscillators and updating the parameters until the entire table has been traversed.

The Pyramid Algorithm

- Look at the (page number part of the) address
- Look for low-order zeroes by doing LSR's
- Count the number of low-order zeroes you shift out
- That number is the value of the Table Size parameter. (Table Size = 0 corresponds to a wave size of one page or 256 bytes.)
- The first segment is the largest wave table that can start on that boundary
- The second segment starting address is (addr) + (wave size #1)
- The third segment starting address is (addr) + (wave size #1) + (wave size #2)

A Small Synthesizer

To summarize what we've learned about the DOC, Bill Mauchly described a typical small synthesizer program built around it. Figure 5 is a block diagram of such a synthesizer. The performance parameters it accepts are:

- Note on/ Velocity
- Note off
- Note on/ Duration (alternative to Note on + Note off)
- Pitch bend ($00-$7F, $40 = none)
- Mod wheel (vibrato amount) ($00-$7F)

Figure 5. A Small Synthesizer

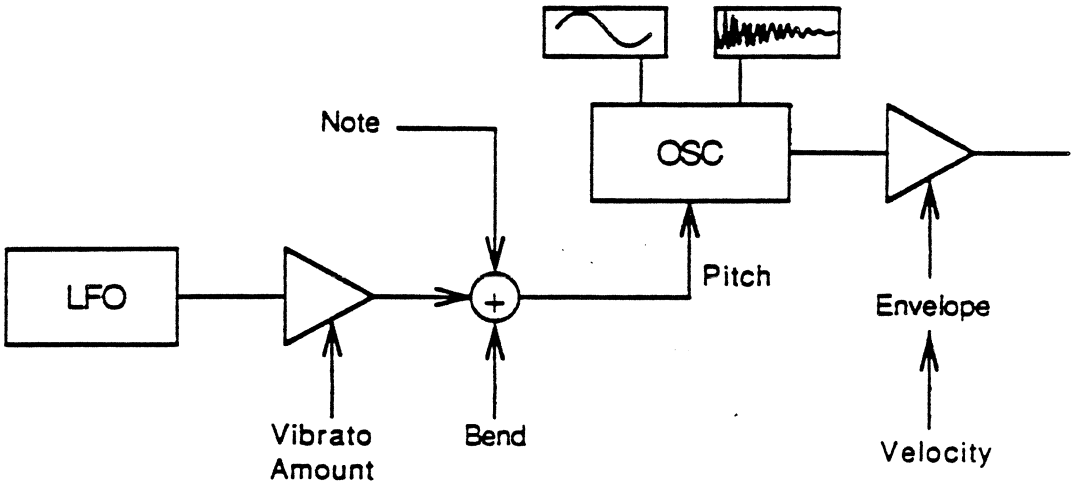


Table 2. Program Control Block ("Patch")

Vibrato Rate

Envelope Shape

ADSR (or arbitrary break points)

Wave I.D. (pointer to segment list)

Chain (pointer to another P.C.B.) ((start same time, to "layer"))

(...envelope for pitch?)

Table 3. Segment List

Wave Address 1

Size 1

Resolution 1

Wave Address 2

Size 2

Resolution 2

(...end?)

Table 4. Voice Control Block

Env. Current Level

Env. Increment

Env. Breakpoint

LFO Phase

LFO Increment

LFO Amoutn

Pitch

The Program Control Block, or patch, contains permanent information about the sound. The Voice Control Block contains information that changes during each note. If the synthesizer program updates the state of the sound every 5 ms, the action list will include things like:

LFO Current = LFO Current + LFO Increment

Env. Current = Env. Current + Env. Increment

IF Env. Stage past Breakpoint THEN get next Inc. and Breakpoint

The synthesizer program generates envelopes by adding increments. To use envelope parameters specified as the length of time it takes for the envelope to rise or fall to the desired level, you have to convert times to rates.

The synthesizer program computes the amplitude levels as linear slopes, then converts them to logarithmic values just before sending them to the VCA.

Envelope times are specified as log(time), that is, a value of 1 for 1 second, 2 for 2 seconds, 3 for 4 seconds, and so on. To change envelope parameters in response to different velocity information from the keyboard, the synthesizer program needs to be able to re-compute increment values without using division, which takes too long. The program uses a table to get logartithms, then subtracts logarithms to perform the necessary division.

To obtain increment (rate) values, you have to divide distance by time, like this:

distance ÷ time = increment

The program does this using logs, like this:

log(distance) - log(time) = log(increment)