Sound & Music

Whether it's commercial programs (most notably in games) or those made by indie developers who may be less enlightened, you've witnessed the musical and sonic capabilities of your Apple IIGS. I'm sure you were not disappointed with the fantastic possibilities, due in large part to having it's own microprocessor for sound, the Ensoniq. However, it's curious that few people have attacked the sound system, which isn't as difficult as it might seem. Once the methods are acquired, sound programming on the Apple IIGS is no longer routine, and it won't get boring.

However, in order to take maximum advantage of the sound potential of your favorite Apple IIGS, you shouldn't settle for the internal loudspeaker. The best way is to connect the IIGS to a HIFI amplifier or to buy specially amplified speakers. And, if you succumb to this purchase, while you are buying there are also a stereo cards that have many benefits we will talk about.

6.1 The Sound

Before embarking on fabulous music, you will first need to learn how to program the Ensoniq to play sounds and then create your own sounds. Then you can already enjoy all the possibilities of sound effects.

Fortunately, Apple has endowed the IIGS with the famous Ensoniq 5503 Digital Oscillator Chip microprocessor, which we will now call more commonly the DOC. Apart from being a chip used in several synthesizers, the DOC has the advantage of being able to play several sounds at the same time, without taking machine time. That is, the 65C816 microprocessor remains fully available for other tasks once the commands are executed.

6.1.1 Description of the system

The Ensoniq DOC has very good sound potential since it has 32 independent oscillators. Each of the oscillators can produce a sound, this means that you can have up to 32 different sounds played at the same time! Of course, if you listen to so many sounds, you may not hear anything at all because of non-harmonization. Moreover, each oscillator can operate with an adjustable and independent volume and in four different modes.

It should be noted that all the registers of the Sound GLU and the DOC are 8 bits wide, so all examples of routines are to be considered with an 8-bit accumulator and XY index registers, unless specifically noted.

6.1.1.1 The sound RAM

The DOC has its own 64K RAM. However, this RAM is quite particular because on the one hand it is not addressable by the 65C816 microprocessor, and on the other hand it is not executable. I.e. - even if you store programs there, they will not work under any circumstances. It is also important to know that during a warm start (type Ctrl-Reset) this RAM Sound is not erased. On the other hand, it is initialized with values $80 during a cold start (Ctrl-Apple-Reset or when the Apple is turned on). Since this RAM is not addressable by the microprocessor, you will need to go through the Sound GLU and its switches.

As for the values to be set in RAM Sound, any value from $01 to $FF is a sound component. Thus a sound wave is a sequence of values which will then be converted into an audible signal. The $00 value is reserved to indicate, if necessary, the end of the sound.

As you can imagine, the sound RAM will contain sounds, but they can't just be any size, so the possible lengths are: $100 ; $200 ; $400 ; $800 ; $1000 ; $2000 ; $4000 or $8000 bytes per sound. As you can see, there is no size of 64K possible; However we will see that this is still achievable with a little trick. Also sounds can not start anywhere in RAM Sound. So they must necessarily start at the beginning of memory page of the RAM Sound.

Here is a small table indicating where a sound can start depending on its size:

Size	Start of the sound, beginning on this page of Sound RAM
$100	$00 ; $01 ; $02 ; $03 ; $04 … $FB ; $FC ; $FD ; $FE ; $FF
$200	$00 ; $02 ; $04 ; $06 ; $08 ... $F6 ; $F8 ; $FA ; $FC ; $FE
$400	$00 ; $04 ; $08 ; $0C ; $10 … $EC ; $F0 ; $F4 ; $F8 ; $FC
$800	$00 ; $08 ; $10 ; $18 ; $20 … $D8 ; $E0 ; $E8 ; $F0 ; $F8
$1000	$00 ; $10 ; $20 ; $30 ; $40 … $B0 ; $C0 ; $D0 ; $E0 ; $F0
$2000	$00 ; $20 ; $40 ; $60 ; $80 ; $A0 ; $C0 ; $E0
$4000	$00 ; $40 ; $80 ; $C0
$8000	$00 ; $80

For example, a sound that is $4000 long (16384 bytes) can start in sound RAM at addresses $0000; $4000; $8000; $C000. But it can not start at $0003, $1000 or $C500.

So you'll notice that the longer a sound is, the fewer places there are to fit it. But what happens

if the size of a sound does not correspond exactly to the standard lengths? In this case, we must pad the excess space with zeros.

For example, if your sound has a size of 13600 bytes long ($3520), it will be considered as 16384 bytes ($4000); But from the 13600th byte until 16383th, you will have to put zeros. Obviously the 2784 bytes set to $00 are wasted, unless you can get another small sound of 2048 bytes. So in this example, both sounds could occupy the RAM sound as follows:

$0000: beginning of the first sound of 13600 bytes declared $4000 long
$351F: end of the first sound
$3520: beginning of the $00 zone indicating the end of the first sound.
$37FF: end of $00
$3800 zone: start of second sound of 2048 bytes declared $800 long
$3FFF: end of second sound

Anyway, the $00 area still represents 736 bytes lost where you could possibly install a sound of 512 bytes. The organized setting up of sounds can really become an art! It is therefore preferable to know exactly the size of all the sounds you want to use before starting the programming.

Despite its small size and its relatively odd organization, the sound RAM is often sufficient for most sound effects or music, if one chooses the sounds well.

6.1.1.2 The control registers (Sound GLU)

The DOC has its own control registers that allow it to be fully programmed. In all there are 227 registers. But to access these registers, you have to go through the four soft switches of Sound GLU (General Logic Unit). This circuit is therefore an intermediary between the Ensoniq DOC and the 65C816 microprocessor. Here are the four control registers of the Ensoniq DOC microprocessor.

GLU Registers	Address	Type
Sound Control	$C03C	read/write
Data Register	$C03D	read/write
Address Low	$C03E	read/write
Address High	$C03F	read/write

Note that the switches can also be accessed using long addressing (example: $E1C03C or $E0C03C), but it is not advised to access them using indexed or other addressing modes. Access to the Sound RAM is also via these registers.

6.1.1.2/1 The Sound Control register at $C03C

This switch controls the overall volume of the internal speaker, but also contains some information about DOC access. Here is the description of its bits:

Bit	Value	Description
7	1	The DOC is busy. Wait until it is free.
7	0	The DOC is free.
6	1	All accesses are in the direction of the Sound RAM
6	0	All accesses are made to the DOC registers
5	1	Auto-increment of register pointers is active
5	0	Auto-increment inactive
4	-	Reserved, do not modify
3-0	$0-$F	Volume control: $0 soft, $F loud

The bit 7 test of this register is normally not necessary, as the Ensoniq is quite fast and rarely used to maximum efficiency. This bit is read only, but if you write in it, nothing will happen.

With respect to bits 5 and 6, if you want to access the DOC registers, I recommend you put them both at zero. On the other hand, when transferring sounds from the main memory to the Sound RAM, put them both to one.

Bit 4 is reserved.

Finally, bits 0 to 3 contain the general volume of the internal speaker of the GS, and of the jack located at the back. However, it should be noted that this volume has no effect in the case of a stereo card because the connectors directly connected to the DOC (on the internal 7-pin Molex connector) have a volume independent of that of the small speaker.

If you want to set the Control Panel volume as the general volume, be aware that the memory $E100CA contains the volume in bits 0 to 3.

Example of accessing DOC registers:

```
            LDAL  $E100CA     ; Read the control panel volume
            AND   #$0F        ; We only keep bits 0 to 3
            STA   $C03C       ; Bits 5 and 6 are forced to zero,
                              ; now we are set to access DOC registers
```

Example of accessing the Sound RAM:

```
            LDAL  $E100CA     ; Read the control panel volume
            AND   #%00001111  ; We only keep bits 0 to 3
            ORA   #%01100000  ; Bits 5 and 6 are forced to one
            STA   $C03C       ; Therefore it is now accessing sound RAM
```

6.1.1.2/2 The data transfer register at $C03D

Whether you're accessing the DOC or the Sound RAM, you'll need to pass data, and that's just the $C03D switch. However, this switch does not work the same way depending on whether you are reading or writing.

In the case of a writing, it suffices to simply issue an STA $C03D.

Example:

```
            LDA   #$xx        ; xx being any hexadecimal value
            STA   $C03D       ; then transfer value to a DOC register
                              ; or into the memory of the Sound RAM
```

But when reading, the behavior of $C03D is different, so if you want to read data, the first byte should not be taken into account if you have just programmed the pointer registers ($C03E or $C03F). We will see examples in the following paragraph.

Moreover, in the case of a reading of a DOC register, the problem is a little more complex because you will often have to change the $C03E switch, so I strongly advise you to systematically perform two readings to read a value!

6.1.1.2/3 The pointer registers at $C03E and $C03F

The two switches $C03E and $C03F are used to indicate where the data will be transferred. In the case of access to the DOC, only the $C03E switch is used. Simply write to the switch which DOC register you want to access. There is no need to deal with the $C03F switch when accessing the DOC since its value is not taken into account.

Example of access to a DOC register:

```
            LDAL   $E100CA     ; First, indicate that you want to access
            AND    #$0F        ; the DOC registers
            STA    $C03C
            LDA    #$xx        ; xx being a hexadecimal number from $00
                               ; to E2 (inclusive) indicating the number
                               ; of the DOC register
            STA    $C03E       ; Which is stored in $ C03E
```

At this point, the DOC register xx is ready to be read or written.

Here is another routine for transferring data between memory and the DOC:

```
*============================================
* Writing to the first 224 registers of the DOC
*============================================
BUFFER      =      $2000       ; Address of the buffer (or elsewhere)
            ORG    $1000       ; The program starts at this address
                               ; (Or another bank $00 address)
            CLC                ; Native Mode
            XCE                ;
            SEP    #$30        ; A and X/Y set to 8 bits
            LDAL   $E100CA     ; Read the Control Panel volume
```

```
            AND   #$0F         ; Bit 4 to 7 to select the mode
            STA   $C03C        ; Access to DOC and without auto-incrementing
            LDX   #00          ; We want to write from start of the buffer
ENCORE      STX   $C03E        ; $C03E contains the register number to write
            LDA   BUFFER,X     ; Reading data from buffer (here $2000)
            STA   $C03D        ; Transfer of data to the DOC
            INX                ; Next value
            CPX   #$E0;        ; Until 224 values were transferred
            BCC   ENCORE
            RTS                ; End of the routine
```

This routine allows you to program the DOC registers in a single transfer, but is only really useful for testing. You will notice that only the first 224 registers of the DOC are concerned, the last three registers being special.

In the same way, here is the reverse routine that allows you to read the registers of the DOC:

```
*=============================================
* Reading the first 224 registers of the DOC
*=============================================
BUFFER      =     $2000        ; Address of the buffer (or elsewhere)
            ORG   $1000        ; The program starts at this address
                               ; (Or another bank $00 address)
            CLC                ; Native Mode
            XCE                ;
            SEP   #$30         ; A and X/Y set to 8 bits
            LDAL  $E100CA      ; Read the Control Panel volume
            AND   #$0F         ; Bit 4 to 7 to select the mode
            STA   $C03C        ; Access to DOC and without auto-incrementing
            LDX   #00          ; We want to write from start of the buffer
```

```
ENCORE      STX   $C03E       ; $ C03E contains the register to be read
            LDA   $C03D       ; Attention, it's necessary to read two
            LDA   $C03D       ; times to avoid problems!
            STA   BUFFER,X    ; Storing the value read in the buffer
            INX               ; Next value
            CPX   #$E0        ; 224 values read?
BCC         ENCORE            ; If so, we have finished
            RTS               ; End of routine
```

On the other hand, when accessing the Sound RAM, the use of the registers $C03E and $C03F together constitute the 16-bit address to which the data must be transferred. Thus $C03E contains the lower part of the address while $C03F contains the upper part.

Moreover, these two registers have the possibility of being auto-incremented whenever data is read or written by $C03D. This is especially useful when you want to transfer data to Sound RAM.

Here is an example of a routine of transferring 64K of sounds from bank $03 to the Sound RAM. Most of the time, it is a routine of this kind that is used to install all sounds at once. Of course, you must preload the desired sounds in bank $03.

```
*============================================
* Write 64K of data to Sound RAM
*============================================
MEMORY      =     $030000      ; Bank where we find the data to write
            ORG   $1000        ; The program starts at this address
                               ; (Or another bank $00 address)
            CLC                ; Native Mode
            XCE                ;
            REP   #$30         ; A and X/Y set to 16 bits
            SEP   #$20         ; And A back to 8 bits.
```

```
            LDAL  $E100CA       ; We chose to access the Sound RAM with
            AND   #%00001111    ; Auto-incrementing pointers ...
            ORA   #%01100000
            STA   $C03C
            STZ   $C03E         ; Zeroing of the two pointers.
            STZ   $C03F
ENCORE      LDAL  MEMORY,X      ; Read a value from bank (here bank $03)
            STA   $C03D         ; Transfer to Sound RAM Sound. Note that at
                                ; the same time the pointers are increased
                                ; by one, therefore it's unnecessary to
                                ; watch over these pointers
            INX                 ; Next value
BNE         ENCORE              ; Until the 64K is transferred
            RTS                 ; Exit the subroutine.
```

Also here is a routing to copy the 64K of Sound RAM to bank $03. It is more rare to read Sound RAM than to write, but this can sometimes be useful and the routine is slightly different from that of writing.

```
*=============================================
* Write 64K of Sound RAM to Bank $03
*=============================================
MEMORY      =     $030000       ; Bank where we find the data to write
            ORG   $1000         ; The program starts at this address
                                ; (Or another bank $00 address)
            CLC                 ; Native Mode
            XCE                 ;
            REP   #$30          ; A and X/Y set to 16 bits
```

```
            SEP   #$20          ; And A back to 8 bits.
            LDAL  $E100CA       ; We chose to access the Sound RAM with
            AND   #%00001111    ; Auto-incrementing pointers ...
            ORA   #%01100000
            STA   $C03C
            STZ   $C03E         ; Zeroing of the two pointers.
            STZ   $C03F
            LDA   $C03D         ; Warning! Here we make a first reading
                                ; for nothing, as shown. But above all,
                                ; it was done after the declaration of
                                ; two pointers and not before!
            LDX   #$0000
ENCORE      LDA   $C03D         ; Reading a value and incrementing
                                ; Automatic pointers.
            STAL  MEMORY,X      ; Storage...
            INX                 ; Next value
            BNE   ENCORE
            RTS                 ; End of the routine.
```

The 32 DOC oscillators are programmed via its 227 registers. Three of these registers relate to the DOC in general, but the remainder are divided into seven sets of 32 registers. Each oscillator can produce one and only one sound at a time, but Apple recommends that oscillators 31 and 32 not be used because they are reserved. However, if you do not use Sound Tools, you can program them without any problems. So you can have up to 32 different sounds at a time.

To effectively produce a sound, you must choose a free oscillator, and indicate in its registers the following:

- The frequency at which the sound will be played.
- The volume of the oscillator.
- The beginning of the sound in Sound RAM.

- The length of the sound.
- The mode of operation of the oscillator and its order of activation.

In the examples in the following paragraphs, the sound control register ($C03C) shall be considered to be programmed to access the records of the DOC. I remind you of the instructions for accessing the DOC:

```
            CLC
            XCE
            REP   #$30        ; A and X/Y set to 16 bits
            SEP   #$20        ; And A back to 8 bits.
            LDAL  $E100CA     ; Read the Control Panel volume
            AND   #$0F        ; Reset bits 5 and 6 to access the DOC
            STA   $C03C       ; Without auto-increment
```

6.1.1.3/1 Frequency registers (low and high) $00 - $3F

What determines the pitch of the sound produced is the frequency at which the sound is played. At low frequency, the sound will be rather deep while high frequency it will tend to be sharp. However, the frequency of the sound that is indicated to the oscillator does not correspond to the actual frequency (in hertz) at which the sound is heard. The programming frequency corresponds to the playback speed of the sound in Sound RAM. Indeed, the oscillator in itself does not produce sound, but it converts each value into a voltage that has the effect of making the speaker membrane more or less vibrate. So converting various values into a sequence will produce a sound. Of course, the reality is more complicated than that, but we will limit ourselves to the programming of the sound without going into the scientific concepts.

The frequency of the sound must be given on 16 bits, which allows to have 65536 different possible frequencies. Obviously, from a certain high frequency, the differences are no longer so audible. However, it seems impossible to reach ultrasound.

The registers $00 to $1F contain the low value of the frequency of each of the 32 oscillators while the registers $20 to $3F the high value of the frequency.

A small detail: if you program an oscillator with a very low frequency ($0001 for example), the sound will play very slowly and will barely audible; however, if you choose a null frequency ($0000), the sound will not be played at all, but the oscillator will not be stopped. This can be used to temporarily interrupt a sound, which can then continue to play from where it was cut.

Example: To set the frequency $156 (342) in the oscillator #$07:

```
LDA   #$07        ; Register $07 is the frequency (low) ...
STA   $C03E       ; of the oscillator #$07.
LDA   #$56        ; Low part of *desired* frequency.
STA   $C03D       ; It is stored via the data register.
LDA   #$27        ; Register $27 is the frequency (high) ...
STA   $C03E       ; of the oscillator #$07.
LDA   #$01        ; High part of the frequency.
STA   $C03D       ; Storage in the DOC.
```

It is also possible to read, if necessary, the frequency registers:

```
LDA   #$07        ; Frequency (low) of the oscillator $07...
STA   $C03E       ; that is indicated in the lower pointer.
LDA   $C03D       ; Two readings are required to read a
LDA   $C03D       ; value, do not forget it!
```

6.1.1.3/2 The Volume registers $40 - $5F

There are 256 possible volumes; moreover, the fact that each oscillator has its own volume becomes very interesting, especially for making music. The volume is increasing: $00 is equivalent to a blank sound, since it is inaudible and $FF is the maximum volume.

Example: Set the volume of oscillator $00 to maximum:

```
LDA   #$40        ; Volume register of oscillator $00.
STA   $C03E
LDA   #$FF        ; $ FF is the maximum value for the volume
STA   $C03D
```

6.1.1.3/3 The Data registers $60 - $7F

These registers are read only. They contain the last value of the Sound RAM read by such and such an oscillator, and their usefulness is not that great. However, we will see in another part a roundabout way to use these registers.

Example: What is the last value read by the oscillator number $1F?

```
            LDA   #$7F        ; Data register of the oscillator $1F.
            STA   $C03E
            LDA   $C03D       ; $ FF is the maximum value for the volume
            LDA   $C03D       ; The accumulator contains the desired value
```

6.1.1.3/4 The address pointer registers to Sound RAM $80 - $9F

To indicate the beginning of the sound used in Sound RAM by an oscillator, it is sufficient to write the beginning of the page to the corresponding register. Refer to the section on Sound RAM to find out where to put the sounds.

Example: We want to indicate the location of the sound to be played by the oscillator $10. Let's take a sound of $4000 bytes long (16384 bytes). It can be installed in Sound RAM at the following addresses: $0000; $4000; $8000; $C000. For this example we will chose the location at $C000:

```
            LDA   #$90        ; Address pointer for oscillator $10.
            STA   $C03E
            LDA   #$FF        ; Page $C0 of the Sound RAM (address $C000)
            LDA   $C03D       ; The accumulator contains the desired value
```

6.1.1.3/5 The sound control registers $A0 - $BF

These are the registers that contain the most information. Here is the description of the bits of one of these registers:

Bits 7 to 4 contain the number of the output channel of the sound currently being played by the oscillator in question, which allows a stereo card to hear sounds on the left or right speakers. For now only stereo cards exist, but it would be quite possible to make cards with 4, 6 or 8

connected speakers, thus allowing extraordinary effects ... However, although we can program up to 16 output channels, only 8 are actually accessible since there are only three of the four "channel address" pins actually connected on the J25 connector.

But anyway, as the stereo card is the de facto standard, it will be enough to turn switch bit 4, the other three remaining to zero. A right output is obtained if bit 4 is at 0 and left if it is at 1. If the oscillator reaches the end of a sound and bit 3 of its control register is set to 1, it will send an interrupt signal to the microprocessor. Of course this signal is handleable and it is even possible to know which oscillator has sent it.

The operating modes of the oscillators determine how the sound will be played. Bits 2 and 1 switch the different modes. The "Free-run" mode produces a sound that once finished will repeat itself indefinitely from the beginning. Conversely, the "One-shot" mode will stop the oscillator at the end of the sound. The two other modes have a more particular use, which we will discuss in a special section devoted to the four modes.

Finally, bit 0 is the oscillator's ON indicator. If it is set to 0, it means that the oscillator is producing a sound, while at 1 it is stopped and therefore produces no sound. Thus, it is sufficient to program this bit to put the oscillator into action, but it must also be known that the switching of this bit can be done automatically according to the modes of operation. Thus, with the "One-shot" mode when the sound has been played in its entirety, the DOC automatically stops the oscillator and bit 0 of its control register is set to 1.

Example: We want oscillator $05 to play a unique sound on the left speaker without producing a break:

1. To play on the left loudspeaker, put $1 in bits 7 to 4 of the $A5 register (0001 in binary).
2. Bit 3 must be set to 0 as no interrupts are to be generated.
3. For a single sound, "One-shot" mode is required, so bits 2 and 1 must contain 0 and 1.
4. To switch on the oscillator, 0 must be set for bit 0.

Thus, the binary value 00010010 ($12) must be set in register $A5.

```
            LDA   #$A5        ; Control register for oscillator  $05
            STA   $C03E
            LDA   #$FF        ; Its on the left, without interrupts.
            LDA   $C03D
```

6.1.1.3/6 The waveform registers $C0 - $DF

We still have to specify the size of the sound used by the desired oscillator. It is among other things the role of the $C0 registers, which the description follows:

Bit 7 is reserved and setting it to zero will be fine. On the other hand, bit 6 must absolutely be set to zero, because it is reserved for a possible extension of the Sound RAM. It is possible that a future IIGS will use 128K of Sound RAM and in this case bit 6 would have the role of switching between the two banks. So, in order not to have a bad surprise of incompatibility with the "next GS", stay in the first bank by setting this bit to zero.

The size of the sound is therefore coded with bits 5 to 3, no particular problem if we've chosen well the size of the sounds and their locations.

The addressing resolution which is encoded with bits 2 to 0 is of little use. It determines according to its coding and according to the size of the sound the number of bytes actually taken into account during the reading of the Sound RAM by an oscillator. Thus at equal frequency and resolution a sound of different size will not have the same height.

Fortunately, choosing the resolution proportionally to the size of the sound cancels out this mostly undesirable effect. In order to help you calculate the values to put in the sound size register, here is a small table indicating the correct values according to the different sizes of sounds:

Size of the sound		Value to put in the $C0 series of registers	
256	$100	$00	%00000000
512	$200	$09	%00001001
1024	$400	$12	%00010010
2048	$800	$1B	%00011011
4096	$1000	$24	%00100100
8192	$2000	$2D	%00101101
16384	$4000	$36	%00110110
32768	$8000	$3F	%00111111

Example: Given a sound that is $4000 bytes long (16384 bytes). We want to indicate this for oscillator $15.

```
            LDA   #$D5        ; Size (waveform) register for the oscillator
```

```
            STA   $C03E       ;  $C0+$15 = $D5
            LDA   #$36        ; According to the table, use value $36
            LDA   $C03D
```

6.1.1.3/7 The Oscillator Interrupt register $E0

This register, like the next two, does not concern a particular oscillator, but the DOC in general. The register $E0 takes care of sound interrupts by first indicating whether an interrupt has been caused and if it was generated by one of the 32 oscillators. Note that this register is read only.

Here's a description of the different bits:

Bit	Value	Description
7	1	No oscillator caused an interrupt
7	0	One of the 32 oscillators caused the interruption
6	-	Reserved
5-1		Contains the number of the oscillator that generated the interrupt
0	-	Reserved

The interrupt indicator (bit 7) is of interest only if you manage the interrupts yourself. In this case, it lets us know that it is the DOC which caused the interruption and not something else (VGC, Clock, mouse, etc...) Of course, if you decide to manage the interrupts yourself, you will have to do a survey routine to locate the source of the interrupt. But the GS already has an excellent system of interrupt management using different branch vectors. In the case of sound, it is in $E1002C. On the other hand, it may be very useful to know which of the 32 oscillators generated the interrupt, unless one oscillator is expected to do so.Thus, in the event of an interrupt, bits 5 to 1 of the register $E0 contain the number of the oscillator, which will then have to be handled.

Example: Extract from an interrupt handling routine:
Let's suppose that the 65C816 microprocessor has just produced an interrupt and that it has branched into your interrupt management routine. This routine aims to know the origin of the interrupt …

```
            ...               ; Other instructions would be here to check
```

```
            ...                 ; whether it is the VGC, the mouse or other
            ...                 ; source of our interrupt

            LDA   #$E0          ; Since neither the VGC nor mouse have
            STA   $C03E         ; produced the interrupt, perhaps it's
            LDA   $C03D         ; it's the DOC?
            LDA   $C03D         ; Therefore let's read the $E0 register, but
                                ; don't forget we have to do two readings!
            BPL   INTERSON
            ...
*====================================
* Sound Interrupt Handler
*====================================
INTERSON    AND   #%00111110    ; Only bits 5 to 1 are kept.
            LSR                 ; A small shift to the right and
                                ; the accumulator now contains the
                                ; number of the oscillator that caused
                                ; the interrupt
            ...                 ; code continues here
```

6.1.1.3/8 The Oscillator Enable register $E1

Normally the 32 oscillators are enabled and each one takes 1.2 microseconds in machine time, which makes about 38 microseconds for the 32. This time is negligible, but if for some reason you would like to go faster, it is possible to select only a part of the oscillators. However, at least one oscillator must be in operation, and it is only possible to activate them in a sequential order. To choose the number of oscillators you want, simply multiply this number by two and indicate it via the register $E1.

However, having fewer oscillators at the same time causes an acceleration of the processing, and the actual sound frequencies are also modified. So I absolutely recommend keeping 32 active oscillators as the time saving is not worth it.

The 32 oscillators are enabled automatically when starting the machine or after a reset, but it is more prudent to indicate this before starting to use sound (it must always be thought that one day it might no longer be compatible).

Here are the necessary instructions:

```
            LDA   #$E1        ; Register $E1
            STA   $C03E
            LDA   #$40        ; We want 32 oscillators, but we multiply
            LDA   $C03D       ; this number by 2, hence $ 40 (64)
```

6.1.1.3/9 The A/D Converter register $E2

The Ensoniq is equipped with a digitizing system that allows the acquisition of analog data and converts it into numbers from 0 to 255. Of course this system is used primarily for the digitization of sound from a tape recorder, for example, but nothing prevents it from being used for other types of scans. To digitize data, simply connect two wires to the DOC connector, one to ground (pin #2) and the other to the input of the converter (pin #1), all connected to the source. However, the maximum voltage allowed during data acquisition is 2.5 volts with a maximum impedance of 3000 ohms. You are strongly advised NOT TO EXCEED THESE VALUES so as not to put the Ensoniq, or even your IIGS, at risk. Either way, if you decide to use digitization by tinkering yourself, it will be at your own risk. Also, if you want to digitize sound and work with care and ease, you better buy a stereo-digitizing expansion card.

As for the scanning routine, be aware that the DOC takes 31 microseconds to convert a value and so you will have to wait for enough time between each acquisition cycle, otherwise you will miss values. If you want to work on specific times, I advise you to switch the speed of the IIGS to 1Mhz, which will facilitate the calculation of the times.

Last point: this register is read only, as one would have suspected!

Here is a sample routine that is intended to sample 64K of data.

```
*-------------------------------------------------------------------
* Example of a scanning routine
```

```
*=====================================================================
MEMORY      =       $030000     ; Data storage bank (you can change)
            ORG     $1000       ; The program starts at this address
                                ; (Or another address if you prefer)
            CLC                 ; Native Mode
            XCE                 ;
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; A back to 8 bits.

            LDA     #$80        ; Set GS speed to 1Mhz by setting
            TRB     $C036       ; setting bit 7 of switch $C036 to zero
            LDAL    $E100CA     ; Volume according to the beep.
            AND     #$0F
            STA     $C03C       ; Access to DOC registers
            LDA     #$E2        ; Choosing the A/D register
            STA     $C03E
            LDA     $C03D       ; First reading for nothing. Notice that
                                ; we do not apply the rule of two readings
                                ; for a value, given that we do not
                                ; change registers
            LDX     #$0000      ; Beginning of the bank
LOOP        LDA     $C03D       ; Read value that has just been converted
            BNE     ADMITTED    ; If it's a $00 then we do not admit it
            LDA     #$01        ; and is replaced by a $ 01
ADMITTED    STAL    MEMORY,X    ; Storage
            INX                 ; Next value
```

```
            BEQ     DONE        ; If we hit 64K of data, then end
            LDY     #$20        ; Delay loop - adjust according to quality
DELAY       DEY                 ; of digitization that one wants to obtain
            BNE     DELAY
            BRA     LOOP        ; Back to a new cycle
DONE        RTS                 ; End of the routine
```

6.1.1.4 The DOC connector

As the title indicates, here we discuss characteristics of the 7-pin "Molex" connector located on the motherboard just to the right of the DOC and to the left of the memory expansion slot. Pin #1 is nearest the green LED of the IIGS.

"Channel strobe" is at the low level when the channel address is valid. The fact that only three of the four output channels of the DOC are connected to the connector allows only 8 possible combinations. So, although it is possible to program the oscillators on 16 distinct channels, you can actually hear only up to eight! But in any case, most people will be content with stereo which uses only two channels.

6.1.2 Creating Sounds

At this point you are almost capable of starting the first tests, except for a small detail: you do not have any sounds yet and you will have to create them. There are different possibilities for sound creation. However, don't forget that you only have 64K of Sound RAM and that $00 is reserved to indicate if the end of a sound. Here are three methods for building a sound library:

6.1.2.1 Digitized Sounds

The principle of digitization consists of converting analog information into digital data. In the case of sound, a sound source (microphone, radio, CD player, etc.) is connected to the IIGS and the sound pulses are transformed into values. Of course, to make these conversions, you must have the necessary hardware and you will have to buy a digitizer expansion card. However most stereo cards also possess digitization and your investment will quickly prove fruitful. In addition, scanning software is normally supplied with these cards.

The sounds obtained this way have a real origin, but their quality compared to the original depends on several parameters: the material used of course, but also the sampling frequency.

Sampling involves converting a certain number of values over time. So, if the sound you want to scan to an actual duration of 3 seconds, you can choose to take samples for example every 40 microseconds or every 80 microseconds. In the first case, the sound will have a better rendering than in the second case, but it will also be twice as long in memory! Alas digitization is costly in the case of memory, and the 64K of Sound RAM can sometimes be a bit constrictive.

Most of the time, the digitized sounds are played in "One-shot" mode for a single sound, or in "Free-run" mode for rehearsals. But, since digitization presupposes the purchase of equipment, it will be impossible for me to give an example here.

6.1.2.2 Synthesized Sounds

Synthesize sounds are completely different from digitization:

- Sounds created have no origin in reality
- The size of the sounds is very short.
- Most of the time 256 bytes.

The principle of synthesis consists in creating a waveform which will represent a period and which will be played in "Free-run" mode, that is to say looping continuously. Any waveform is imaginable, but some forms are more harmonious than others and only final listening can be a criterion of choice.

For our example, we will create a sinusoidal wave using a small program in Basic:

```
10 FOR X = 0 TO 255
20 Y = INT (128-127*SIN(X/40.6))
30 POKE 16384-1-X,Y
40 NEXT X
```

Line 10: We want to create a wave of 256 bytes.
Line 20: You need integer values. The 128 is used because it corresponds to the middle of the ordinate axis, if we imagine a mark representing the curve. The formula was studied in order not to give a zero value. Remember that a zero would indicate the end of the sound, and we want to play the sound continuously. Remember also that the Basic Applesoft calculates in radians.
Line 30: The wave will be stored from $4000 to $40FF.

Then we will transfer the wave to Sound RAM at addresses $0000 - $00FF. Then you have to program the registers of the DOC to finally hear something. We will use the oscillator $00 for our example, with a frequency of $65B and its maximum volume. But before starting the programming, here is a small table summarizing the exact values to put in the registers:

Register	Value

$00	$5B (low frequency part)
$20	$06 (high frequency part)
$40	$FF (maximum volume)
$80	$00 (page where the sound begins, at $0000 of the Sound RAM)
$A0	$00 (right output channel, no interrupt, free-run mode, oscillator in action)
$C0	$00 (wave takes up 256 bytes)

It is advisable to program the various registers in their order, but it is absolutely necessary to program the control register last, because that will trigger the sound. On the other hand, if it were not declared last, there would certainly be a problem because the sound would be played before its parameters were defined!

Here is a routine in assembler which will play this wave after transferring it to Sound RAM:
MISSING

6.1.2.3 Synthesized Sounds used as digitized sounds

This type of sound is a compromise between digital and synthetic sounds. Indeed, they do not have a real existence, like synthetic sounds, but they are used in the same way as digitized sounds. The advantages are, on the one hand, the purity of the sounds (unlike digitization, which always produces some artifacts), and on the other hand there is no need for any apparatus relating to digitization. However the disadvantage is not being able to obtain the desired sounds and here again it will take a lot of tests to find pleasant sounds. Moreover, since the size of this type of sound is close to that of the digitized sounds and is constructed by calculations, a certain waiting time is required.

For our example, we will create a sound of 16K ($4000 bytes) using a small BASIC program. Beware, the program takes about six minutes to calculate everything.

```
10 FOR X = 0 TO 16383
20 Y = INT (128-1-(121-(X/140))*SIN(.12*X)))
30 POKE 81942-4-X,Y
40 NEXT X
```

Line 20: The formula is designed to give a sound whose volume decreases over time. Of course you are free to create any sound at any time as long as it doesn't contain a null value.
Line 30: The sound is stored from $2000 to $5FFF.

A small "BSAVE SOUND,A$2000,L$4000" would be a good idea for saving the sound on a disk in ProDOS 8 format so as not to have to wait six minutes each time.

Once the sound is created and saved, type the following commands:

```
]CALL -151
0<3/0.FFEEZ             (Set bank $03 to zero)
3/0<0/2000.5FFFM        (transfer sound to beginning of bank $03)
```

The sample program to play a digitized sound would be the same as for this type of sound. Thus we want oscillator $01 (for a change) to play this sound once ("One-shot" mode), at frequency $130, volume $C0, and on the left channel. Here is the summary table:

Register	Value
$01	$30 (low frequency part)
$21	$01 (high frequency part)
$41	$C0 (maximum volume)
$81	$00 (page where the sound begins, at $0000 of the Sound RAM)
$A1	$12 (left output channel, no interrupt, one-shot mode, oscillator in action)
$C1	$36 (the wave takes up 16384 bytes) Refer to the table for section 6.1.1.3/6

As you can see, the programming of a sound is in itself extremely simple, once we know the basics of operation of the Ensoniq DOC.

6.1.3 The 4 operating modes of the DOC oscillators

As we have seen, the oscillators of the DOC can operate in four modes, chosen by programming bits 1 and 2 of the $A0 series of registers. Some of these modes use only one oscillator, others must be grouped in pairs. In the case where they are grouped by two, it is always an even oscillator coupled with its odd neighbor, for example $00 with $01, or $08 with $09, or $1E with $1F, etc …

6.1.3.1 “One-Shot” mode

This is the simplest mode we have used. Mainly used for long sounds (like digitized sounds), it only needs one oscillator to work.

Programming the oscillator with the "one-shot" mode resets its accumulator, i.e. - when the oscillator is switched on, the sound must be played from the beginning. The oscillator will play the sound once and then stop at the end of the sound.

Last detail in the "One-shot" mode: it has no problem playing a sound ending with zeros.

6.1.3.2 “Free-Run” mode

Also used with a single oscillator, this mode allows you to play the same sound (looping) several times. For synthesized sounds, usually short sizes (256 bytes most often), this is the quintessential mode.

However, to replay the same sound several times, it must be an exact size (256, 512, 1024, 2048, 4096, 8192, 16384 or 32768 bytes) without any zeros. If it reaches a zero, the oscillator would stop itself.

Unlike the “one-shot” mode, the "free-run" mode does not reset the oscillator's accumulator to zero, which means that if you interrupt a sound being played by stopping the oscillator, then you turn it back on, the sound will resume where it was and not from the beginning.

6.1.3.3 “Swap” mode

This mode, which operates with a pair of oscillators, offers interesting possibilities. The operating principles are as follows: an oscillator programmed in "swap" mode will play the sound as if it were in "one-shot" mode, then will stop and put into action the neighboring oscillator. But there are different possibilities to program even and odd oscillators in combination with Swap mode:

Oscillators		Effect Sought
Even	Odd	
Swap	Swap	Possible to play the 64K of Sound RAM or a sound ending with zeros, emulating Free-run mode
Swap	One-Shot	Possible to play the 64K of the Sound RAM emulating the mode "One-shot" or play the same sound twice.
One-Shot	Swap	Possible to playing the even oscillator in “free-run” emulation mode. To do this, it is enough to set the odd oscillator to "swap" mode without starting it or to program its other registers except the control register.

The other combinations that would be possible with the Swap mode and Free-run and Sync modes do not do anything interesting.

Example of two oscillators programmed in "Swap" and "Swap" mode to play a sound ending with zeros. The sound will be played by oscillators $06 and $07 alternately, with a frequency of $120 and max volume on the right channel. First, perform the following operations to load your sound:

```
]BLOAD SOUND            (load the sound at $2000)
]CALL -151              (enter the monitor)
3/4000<0/2000.5FFFM     (transfer sound to $034000-$037FFF)
0<3/7500.7FFFM          (here we put a sequence of zeros so the sound
                         occupies only $3500 bytes)
```

So our sound has an actual size of 13568 bytes. However, according to the structure of the Sound RAM, it is necessary to declare its size by excess, hence a size of $4000 bytes and the value $36 to put in the $C0 series of registers. Here is the table summarizing the values put in the registers of the oscillators $06 and $07:

Register	Value
$06	$20 (low frequency part - even oscillator)
$07	$20 (low frequency part - odd oscillator)
$26	$01 (high frequency part - even oscillator)
$27	$01 (high frequency part - odd oscillator)
$46	$FF (volume - even oscillator)
$47	$FF (volume - even oscillator)
$86	$40 (the sound starts at $4000 in sound ram)
$87	$40 (same for the odd oscillator)
$A6	$06 (right output channel ; no interrupts ; “Swap” mode ; oscillator active)
$A7	$07 (right output channel ; no interrupts ; “Swap” mode ; oscillator stopped)
$C6	$36 (our sound is 13568 bytes long which is still declared as $4000, so we use the value $36 as defined in our size table)
$C7	$36 (same for the odd oscillator)

Note that oscillator $06 is going to be the first to go into action, while the $07 is initially stopped. Thereafter, $06 will be stopped as soon as it encounters a $00 in Sound RAM, immediately triggering $07, and so on …

Here is the final program, with the sound transfer routine, which should now be familiar:

```
*=====================================================================
* Routine playing a sound terminated by zeros (first method)
*=====================================================================
MEMORY      =       $030000     ; Data storage bank
            ORG     $1000       ; The program starts at this address
                                ; (Or another address if you prefer)
            CLC                 ; Native Mode
            XCE                 ;

*====================================
* Writing to the 64K of Sound RAM
*====================================
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; And A back to 8 bits.
            LDAL    $E100CA     ; One chooses to access the
            AND     #%00001111  ; RAM Sound with auto-increment
            ORA     #%01100000  ; pointers ...
            STA     $C03C
            STZ     $C03E       ; Resetting the two pointers
            STZ     $C03F
            LDX     #$0000      ; We want to read from start of bank
ENCORE      LDAL    MEMORY,X    ; Read value from other bank ($03 here).
            STA     $C03D       ; Transfer to Sound RAM. At the same time,
                                ; the pointers are increased by one, so
```

```
                                    ; it is useless to manage the pointers.
            INX                     ; Next value.
BNE         ENCORE                  ; Until the 64K is transferred.

*====================================
* Programming the DOC
*====================================
            LDAL    $E100CA     ; Volume according to the beep
            AND     #$0F
            STA     $C03C       ; Access to DOC registers.

            LDA     #$06        ; Frequency register (low) of
            STA     $C03E       ; oscillator $06
            LDA     #$20
            STA     $C03D
            INC     $C03E       ; Register of oscillator $07
            STA     $C03D       ; Note the accumulator still contains $20
            LDA     $26         ; Frequency register (high).
            STA     $C03E
            LDA     #$01
            STA     $C03D
            INC     $C03E
            STA     $C03D
            LDA     #$46        ; Register volume
            STA     $C03E
            LDA     #$FF
```

```
        STA     $C03D
        INC     $C03E
        STA     $C03D
        LDA     #$86        ; Address register in sound RAM.
        STA     $C03E
        LDA     #$40
        STA     $C03D
        INC     $C03E
        STA     $C03D
        LDA     #$C6        ; Sound size register
        STA     $C03E
        LDA     #$36
        STA     $C03D
        INC     $C03E
        STA     $C03D
        LDA     #$A6        ; The control register of the oscillator.
        STA     $C03E       ; Attention, it is always preferable
        LDA     #$06        ; to write the control register last
        STA     $C03D       ; last, because that's what actually
        INC     $C03E       ; triggers the sound.
        LDA     #$07        ; Note that at the beginning only the
        STA     $C03D       ; even oscillator will be in action.
        RTS                 ; Exit. The sound is being played
```

This technique for playing a sound ending in zeros is quite valid, however it has the disadvantage of asking for the same values twice for even and odd registers. So this can be annoying if you need to cut short the sound playing, because you will have to do it for the even oscillator, but also for the odd oscillator. Here is the second method, using two oscillators

programmed in "One-shot" and "Swap" mode. We will keep the same sound and the same parameters. The table of values is as follows:

Register	Value
$06	$20 (low frequency part - even oscillator)
$26	$01 (high frequency part - even oscillator)
$46	$FF (volume - even oscillator)
$86	$40 (the sound starts at $4000 in sound ram)
$A6	$06 (right output channel ; no interrupts ; “One-Shot” mode ; oscillator active)
$A7	$07 (right output channel ; no interrupts ; “Swap” mode ; oscillator stopped)
$C6	$36 (our sound is 13568 bytes long which is still declared as $4000, so we use the value $36 as defined in our size table)

Note that only the fact that the odd oscillator is in "Swap" stopped mode is important. We do not care about the other parameters of the odd oscillator.

Now here’s the routine:

```
*=====================================================================
* Routine playing a sound terminated by zeros (first method)
*=====================================================================
MEMORY      =        $030000     ; Data storage bank
            ORG      $1000       ; The program starts at this address
                                 ; (Or another address if you prefer)
            CLC                  ; Native Mode
            XCE                  ;

*====================================
* Writing to the 64K of Sound RAM
*====================================
```

```
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; And A back to 8 bits.
            LDAL    $E100CA     ; One chooses to access the
            AND     #%00001111  ; RAM Sound with auto-increment
            ORA     #%01100000  ; pointers ...
            STA     $C03C
            STZ     $C03E       ; Resetting the two pointers
            STZ     $C03F
            LDX     #$0000      ; We want to read from start of bank
ENCORE      LDAL    MEMORY,X    ; Read value from other bank ($03 here).
            STA     $C03D       ; Transfer to Sound RAM. At the same time,
                                ; the pointers are increased by one, so
                                ; it is useless to manage the pointers.
            INX                 ; Next value.
BNE         ENCORE              ; Until the 64K is transferred.

*====================================
* Programming the DOC
*====================================
            LDAL    $E100CA     ; Volume according to the beep
            AND     #$0F
            STA     $C03C       ; Access to DOC registers.

            LDA     #$06        ; Frequency register (low) of
            STA     $C03E       ; oscillator $06
            LDA     #$20
```

```
        STA     $C03D
        LDA     #$26        ; Frequency register (high)
        STA     $C03E
        LDA     #$01        ; Frequency register (high).
        STA     $C03D
        LDA     #$46        ; Register volume
        STA     $C03E
        LDA     #$FF
        STA     $C03D
        LDA     #$86        ; Sound RAM address register
        STA     $C03E
        LDA     #$40        ; $4000
        STA     $C03D
        LDA     #$C6        ; Sound size register
        STA     $C03E
        LDA     #$36
        STA     $C03D
        LDA     #$A6        ; Oscillator control register
        STA     $C03E       ; Note: It’s always preferable to
        LDA     #$02        ; write to the control register last,
        STA     $C03D       ; as it effectively launches the sound.
        INC     $C03E
        LDA     #$07        ; Note: The odd oscillator must always
        STA     $C03D       ; be set to "swap" mode and stopped

        RTS                 ; Exit. The sound is being played
```

6.1.3.4 “Sync” mode

The "Sync" mode also uses a pair of oscillators, and generally only the odd oscillator will be programmed with this mode. The effect caused is the synchronization of the odd oscillator on the even one, producing a better sound level. There is no need to program the other registers of the odd oscillator, only its control register must be in "Sync" mode and may even be stopped. It must be admitted that this mode is not of great use, although it is possible to obtain certain effects according to the way of programming the two oscillators. For example, put the two oscillators in "Sync" mode, or only the first one, or choose a different frequency for each one ... the possibilities are too numerous to give an example, and it’s easy for you to test them.

6.1.4 Sound Interrupts

The ease with which interrupts are handled by the IIGS is such that it would be a shame there wasn’t the same functionality for sound. Indeed, each DOC oscillator has the ability to send an interrupt signal if it has been programmed for it. The signal is triggered at the end of the sound. If you use the normal interrupt handling system, the sound connection vector is in $E1002C and you just need to put a long JMP into your processing routine. Subsequently your routine will determine, if necessary, which of the 32 oscillators produced the interrupt, using DOC register $E0 (see Section 6.1.1.3.7 for the use of this register). Of course, if there is only one oscillator programmed to produce interrupts, it will be useless to use register $E0.

Our example will be to change the border color each time the sound is played.

```
]BLOAD SOUND            (load sound at $2000)
]CALL -151 ⌧            (enter the monitor)
3/0<0/2000.5FFFM        (transfer sound into $030000-$033FFF)
```

We will use oscillator $1F, programmed to play the sound continuously on the left channel, with a frequency of $160, a volume set to half and the interrupt authorized.

Here is the table of values:

Register	Value
$1F	$60 (low part of frequency)
$3F	$01 (high part of frequency)
$5F	$80 (volume = 50%)
$9F	$00 (sound starts at $0000 in Sound RAM)

$BF	$18 (left output channel ; interrupts ON ; “free-run” mode, ; oscillator active)
$DF	$36 (our sound is $4000 bytes long)

```
*=====================================================================
* Routine playing a sound producing interrupts
*=====================================================================
MEMORY      =       $030000     ; Data storage bank
            ORG     $1000       ; The program starts at this address
                                ; (Or another address if you prefer)
            CLC                 ; Native Mode
            XCE                 ;

*====================================
* Writing to the 64K of Sound RAM
*====================================
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; And A back to 8 bits.
            LDAL    $E100CA     ; One chooses to access the
            AND     #%00001111  ; RAM Sound with auto-increment
            ORA     #%01100000  ; pointers ...
            STA     $C03C
            STZ     $C03E       ; Resetting the two pointers
            STZ     $C03F
            LDX     #$0000      ; We want to read from start of bank
ENCORE      LDAL    MEMORY,X    ; Read value from other bank ($03 here).
```

```
            STA     $C03D       ; Transfer to Sound RAM. At the same time,
                                ; the pointers are increased by one, so
                                ; it is useless to manage the pointers.
            INX                 ; Next value.
BNE         ENCORE              ; Until the 64K is transferred.

*====================================
* Divert the sound vector to our
* own handler routine
*====================================
            LDA     #$5C        ; Long JMP code.
            STAL    $E1002C     ; $E1002C is the sound vector.
            LDA     #<INTER     ; Lower part of the address of our routine
            STAL    $E1002D     ; to handle sound interrupts.
            LDA     #>INTER     ; High part of the address.
            STAL    $E1002E
            LDA     #$00        ; We are in bank $00.
            STAL    $E1002F

*====================================
* Programming the DOC
*====================================
            LDAL    #$E100CA
            AND     #$0F
            STA     $C03C       ; Access to DOC registers.
            LDA     #$06        ; Frequency register (low) of
```

```
        STA     $C03E       ; oscillator $1F
        LDA     #$60
        STA     $C03D
        LDA     #$3F        ; Frequency register (high)
        STA     $C03E
        LDA     #$01
        STA     $C03D
        LDA     #$5F        ; Volume register
        STA     $C03E
        LDA     #$80
        STA     $C03D
        LDA     #$9F        ; Sound RAM address register
        STA     $C03E
        LDA     #$00
        STA     $C03D
        LDA     #$DF        ; Sound size register.
        STA     $C03E
        LDA     #$36
        STA     $C03D
        LDA     #$BF        ; The oscillator control register.
        STA     $CO3E       ; Note: it is always preferable to
        LDA     #$18        ; write the control register last,
        STA     $C03D       ; because that's what will actually
                            ; trigger the sound.
        RTS                 ; Exit. The sound is being played and the
                            ; border color changes at the end of the
```

```
                                ; sound.
INTER       SEP     #$30        ; A,X,Y all set to 8 bits
            INC     $C034       ; Change the border color
            CLC
            RTL                 ; End of the interrupt handler routine
```

This routine, which is called automatically, is only a simple example, but you can already see the possibilities. Changing the border color is hardly interesting, but you will notice that changing the oscillator frequency will also change the speed of color variation. Indeed, this routine is nothing but a tempo, which we will see is a very useful trait for music.

6.1.5 Stereo

If outside the memory extension there was only one card to buy, it would be a stereo card! Indeed, the IIGS and its sound capabilities really deserve it and you won't regret adding some speakers either. Thus such a card allows each oscillator to choose whether the sound will come out to the right or left. From the programming point of view, each oscillator programmed with an even channel will play to the right, while with an odd channel it will be left.

However it is better to choose the zero channel for the right and the one for the left.

Besides stereo this expansion card offers other advantages. Already, connecting speakers to the output of the board and not to the output of the IIGS avoids some noise, because the sound is transmitted only through the interface of the Ensoniq. As a result, the control-G beep will not be reproduced by the speakers connected to the stereo card. This is very important because you will be able to lower the beep volume level so as not to be disturbed, without losing the volume of the GS sounds, since the overall volume control in the control panel only affects the built-in internal speaker and the jack on the back of the IIGS.

Most stereo cards also feature a sampling system to digitize sounds, and you would be wrong to deprive yourself of it. In addition, some cards have their own adjustable amplifier. But if you want to buy a stereo card, you should also buy some speakers, preferably amplified, unless you want to connect to your HIFI system via the card.

6.1.6 Special Effects and Miscellanea

This section contains some examples of what can be done with the IIGS and the sound, but it is not exhaustive.

6.1.6.1 Simple Echo

The method of producing an echo is to play a sound initially at high volume, then a second time but less strong. But our interest is to do it automatically, without the intervention of a program doing the processing. For this, it is enough to use two oscillators, the first one programmed in "Swap" mode and active at maximum volume, and the second one in "One-Shot" mode stopped with a volume lower. Of course, the frequency of the two oscillators must be the same, unless you want another effect. Given the simplicity of the routine, it's not beneficial to provide a code example here.

6.1.6.2 Panning right-left

The effect of passing a sound from one speaker to another while it is being played is quite interesting, but it is necessary that a program intervenes during processing and you will not be able to do anything else during this time.

The principle is to use two oscillators programmed in "One-shot" mode which will play the sound at the same time, one on the right high volume, the other on the left at zero volume. But as the sound is played, a routine will decrease the volume of the first oscillator while increasing the volume of the second. This creates a pleasant effect, provided that both speakers are well arranged and of course you must have a stereo card, without which there would be no effect.

For our example, we'll load our sound as usual, and play it at frequency of $120.

```
]BLOAD SOUND            (load sound at $2000)
]CALL -151 ⌧            (enter the monitor)
3/0<0/2000.5FFFM        (transfer sound into $030000-$033FFF)
```

We will use the oscillators $00 and $01, here is the table of values:

Register	Value
$00	$20 (low part of frequency - even oscillator)
$01	$20 (low part of frequency - odd oscillator)
$20	$01 (high part of frequency - even oscillator)
$21	$01 (high part of frequency - odd oscillator)
$40	$FF (volume - even oscillator)
$41	$00 (volume - odd oscillator)
$80	$00 (sound starts at $0000 in Sound RAM)

$81	$00 (same for the odd oscillator)
$A0	$02 (right output channel ; no interrupts ; “One-shot” mode ; oscillator active)
$A1	$12 (left output channel ; no interrupts ; “One-shot” mode ; oscillator active)
$C0	$36 (our sound is $4000 bytes long)
$C1	$36 (same for the odd oscillator)

```
*=======================================================================
* Routine playing a sound from right to left
*=======================================================================
MEMORY      =       $030000     ; Data storage bank
            ORG     $1000       ; The program starts at this address
                                ; (Or another address if you prefer)
            CLC                 ; Native Mode
            XCE                 ;

*====================================
* Writing to the 64K of Sound RAM
*====================================
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; And A back to 8 bits.
            LDAL    $E100CA     ; One chooses to access the
            AND     #%00001111  ; RAM Sound with auto-increment
```

```
            ORA     #%01100000  ; pointers ...
            STA     $C03C
            STZ     $C03E       ; Resetting the two pointers
            STZ     $C03F
            LDX     #$0000      ; We want to read from start of bank
ENCORE      LDAL    MEMORY,X    ; Read value from other bank ($03 here).
            STA     $C03D       ; Transfer to Sound RAM. At the same time,
                                ; the pointers are increased by one, so
                                ; it is useless to manage the pointers.
            INX                 ; Next value.
BNE         ENCORE              ; Until the 64K is transferred.

*====================================
* Programming the DOC
*====================================

            LDAL    #$E100CA
            AND     #$0F
            STA     $C03C       ; Access to DOC registers.
            LDA     #$00        ; Frequency register (low) of
            STA     $C03E       ; oscillator $00
            LDA     #$20
            STA     $C03D
            INC     $C03E       ; and oscillator $01
            STA     $C03D
            LDA     #$21        ; Frequency register (high) of
```

```
        STA     $C03E       ; oscillator $00
        LDA     #$01
        STA     $C03D
        STA     $C03E       ; and oscillator $01
        STA     $C03D
        LDA     #$40        ; Volume register
        STA     $C03E
        LDA     #$FF        ; oscillator $00 at max volume
        STA     $C03D
        INC     $C03E
        LDA     #$00        ; oscillator $01 at min volume
        STA     $C03D
        LDA     #$80        ; Sound RAM address register
        STA     $C03E
        LDA     #$00
        STA     $C03D
        INC     $C03E
        STA     $C03D
        LDA     #$C0        ; Sound Size register
        STA     $C03E
        LDA     #$36
        STA     $C03D
        INC     $C03E
        STA     $C03D
        LDA     #$A0        ; The oscillator control register.
        STA     $C03E
```

```
            LDA     #$02
            STA     $C03D
            INC     $CO3E
            LDA     #$12        ; At this point the sound is playing,
            STA     $C03D       ; but only audible on the right channel.

*====================================
* Volume Modification
*====================================
            LDY   #$FF        ; We will now "slide" the sound to the left,
                              ; progressively, 255 times
LOOP        LDA   #$40        ; Access right volume
            STA   $C03E
            LDA   $C03D       ; Two reads are necessary
            LDA   $C03D
            DEC               ; Minus one
            STA   $C03D
            LDA   #$41        ; Access left volume
            STA   $C03E
            LDA   $C03D
            LDA   $C03D
            INC               ; But this time, add one
            STA   $C03D
            LDX   #$FF        ; To avoid sliding the sound *too* fast,
LOOP2       JSR   WAITING     ; it is necessary to have a delay routine
            DEX               ; which depends on the length of the sound
```

```
            BNE   LOOP2       ; and the frequency value.
            DEY
            BNE   LOOP
            RTS               ; End of routine.

Waiting     DS    18,$EA      ; In this specific case (length of sound is
                              ; $4000 bytes and frequency of $120), it
                              ; must be 18 NOP (whose code is $EA).
            RTS               ; End of the delay subprogram.
```

Note that this function is set up to distribute the effect of the sliding of the volume from right to left so that the maximum volume arrives on the left speaker exactly at the end of the sound.

6.1.6.3 Offset frequencies

The effect called "offset frequencies" can sometimes be very surprising when you're not expecting it. The principle consists of playing the same sound on several oscillators at the same time, but different frequencies. For example, the first oscillator would play the sound with a frequency of $120, the second with $121, the third with $122, etc. The operating mode of all oscillators would be either "One-shot" or "Free-run". Anyway the effect is spectacular because at first the sounds play at the same time, but very quickly they shift from each other. The larger the number of oscillators, the more striking the result, but be careful not to saturate the speakers by simultaneously lowering the oscillator volume if necessary.

6.1.6.4 Ensoniq and chance

Quite often, programmers need random numbers in their applications, especially if they are games where chance plays an important role. There are a variety of ways to get random numbers, and I'm going to offer you an original method based on a misuse of the $60 series of DOC registers.

In fact, these registers contain the last value read by their corresponding oscillator (see paragraph 6.1.1.3.3) and have no practical use for sound. But now let's imagine that an oscillator plays a 256-byte continuous, zero-volume sound. It would suffice to read its corresponding data register to obtain one of the 256 values contained in the Sound RAM wave. It would be enough to put any values in this wave (read the data register whenever needed to obtain a random number.) Here is the basis of an example:

Program the first 256 bytes of the Sound RAM with a sequence of $01 and $02.

Program oscillator $00 to play the first 256 bytes of the Sound RAM in "Free-run" mode, frequency $99 (preferably choose an odd frequency), volume $00 (because there is no interest in hearing this sound).

When you need a random value (here either a $01 or $02), perform the following instructions:

```
            LDA   #$60        ; Access to the data register
            STA   $C03E
            LDA   $C03D       ; Two reads are necessary
            LDA   $C03D       ; The accumulator contains a random value
```

The advantages of such a routine are: ease, the fact that it is automatic, the ability to configure exactly the numbers you want. Indeed, you can very well, in our example, put a higher percentage of $01 than $02 to favor this or that number. So yes, you can even configure your own probabilities!

Only restriction: it is necessary to read the data register in an irregular way. But in the case of a large program, it is rare to have regular cycles of time.

6.2 Music

Though the Ensoniq is able to automatically manage sounds, it is not the same for music. So you will have to elaborate a routine yourself. The idea is to create a music routine independent of your program, using the interrupts of course. There will be no time-shifting problem related to your program and, once started, the music will be executed with a minimum of machine time. But anyway, making music on an Apple IIGS is only the logical outcome of the sound and the programming remains at a simple level.

6.2.1 The 64K Method

If for any reason you need all your machine time or all the IIGS RAM and you still want to hear music, here is a very simple method. It's just a matter of digitizing 64K of music from a CD or other device, then playing a few seconds continuously so that it feels like hearing an endless song. Of course, you'd better pay attention to choose a really rhythmic passage without words. Alas, after a few moments of listening, even if the result is quite acceptable, the music will nevertheless becomes very tiresome.

So, if you choose this solution, you only need to program two oscillators in "Swap" mode, same frequency for both. The first oscillator playing the first 32 kilobytes of the Sound RAM, while the second one plays the other 32K.

6.2.2 Music and Interrupts

However, if you are looking for a real solution to play music in your programs, you will need to use the interrupts. It is certain that there are several possible methods with interrupts, but that using them for the tempo seems to me the most interesting from the point of view of facility and rapidity of execution.
Here, the sounds representing the instruments will be stored in Sound RAM, and the notes with their durations will be written somewhere in normal RAM. The notes will have to be converted into frequencies according to a given table, while the durations will actually be waiting times between each note change. Unfortunately, it is very difficult to find the true frequencies corresponding to the notes in the scale, since not all sounds have been digitized at the same frequency.

In order to function, the routine must interrupt the current program every *X* time and carry out various changes of notes and duration. Of course, the programmer should make sure that his music routine is as fast as possible so as not to disturb the main program.

Here is an example of a small routine playing music with only one sound. The routine uses the oscillator $00 and $01 alternately to avoid hearing the starting and stopping of the oscillators.

To perform this routine, just load our sound:

```
]BLOAD SOUND            (load sound at $2000)
]CALL -151 ⌧            (enter the monitor)
3/0<0/2000.5FFFM        (transfer sound into $030000-$033FFF)
```

Then just run the routine in $1000.

```
*=====================================================================
* Music Routine
*=====================================================================
MEMORY      =       $030000     ; Data storage bank
FREQBAS     =       $FE         ; Two bytes at page zero
FREQHI      =       $FF         ; used to save frequencies
            ORG     $1000       ; The program starts at this address
```

```
                                ; (Or another address if you prefer)
            CLC                 ; Native Mode
            XCE                 ;

*====================================
* Writing to the 64K of Sound RAM
*====================================
            REP     #$30        ; A,X,Y registers to 16 bits
            SEP     #$20        ; And A back to 8 bits.
            LDAL    $E100CA     ; One chooses to access the
            AND     #%00001111  ; RAM Sound with auto-increment
            ORA     #%01100000  ; pointers ...
            STA     $C03C
            STZ     $C03E       ; Resetting the two pointers
            STZ     $C03F
            LDX     #$0000      ; We want to read from start of bank
ENCORE      LDAL    MEMORY,X    ; Read value from other bank ($03 here).
            STA     $C03D       ; Transfer to Sound RAM. At the same time,
                                ; the pointers are increased by one, so
                                ; it is useless to manage the pointers.
            INX                 ; Next value.
            BNE     ENCORE      ; Until the 64K is transferred.

*====================================
* Rerouting the sound interrupt vector
*====================================
```

```
            SEP     #$30
            LDA     #$5C        ; $5C = JMP Long opcode
            STAL    $E1002C     ; The sound interrupt vector
            LDA     #<INTER     ; is found at $E1002C
            STAL    $E1002D
            LDA     #>INTER
            STAL    $E1002E
            LDA     #$00
            STAL    $E1002F

*====================================
* Setting the tempo
*====================================
            LDA     $E100CA
            AND     #$0F        ; Access DOC registers
            STA     $C03C
            LDA     #$1F        ; Using oscillator $1F (#31) for
            STA     $C03E       ; tempo with a frequency of $100.
            LDA     #$00        ; By increasing the frequency, you
            STA     $C03D       ; play the music faster.
            LDA     #$3F
            STA     $C03E
            LDA     #$01
            STA     $C03D
            LDA     #$5F        ; The volume of this oscillator is at
            LDA     #$00        ; zero because we don't want to hear
```

```
            STA     $C03D       ; the sound of the tempo.
            LDA     #$9F
            STA     $C03E
            LDA     #$00        ; The “sound” played starts at $0000 of
            STA     $C03D       ; the Sound RAM and only has a duration
                                ; of 256 bytes.
            LDA     #$BF
            STA     $C03E
            LDA     #$00
            STA     $C03D
            LDA     #$BF
            STA     $C03E
            LDA     #$08        ; Use the “Free-run” mode
            STA     $C03D

*=======================================
* Fixed values for oscillators $00 & $01
*=======================================
            LDA     #$40        ; The volume of the two oscillators
            STA     $C03E       ; will be set to $50
            LDA     #$50
            STA     $C03D
            INC     $C03E
            STA     $C03D
            LDA     #$80        ; The sound we’re using starts at $0000
            STA     $C03E       ; in Sound RAM.
```

```
            LDA     #$00
            STA     $C03D
            INC     $C03E
            STA     $C03D
            LDA     #$C0        ; The sound has a size of $4000 bytes
            STA     $C03E
            LDA     #$36
            STA     $C03D
            INC     $C03E
            STA     $C03D

*=======================================
* Initialization
*=======================================
            REP     #$30        ; A and X/Y to 16 bits
            STZ     NUMBER      ; Reset counter notes
            STZ     DELAY       ; and time limit
            RTS                 ; Return to caller.  At this instant, the
                                ; music player is launched and functions
                                ; automatically

*=======================================
* Interrupt routine
*=======================================
INTER       REP     #$30        ; At each interrupt caused by
                                ; oscillator $1F (tempo), the IIgs
```

```
                                ; will call this routine.
        LDA     DELAY           ; If the delay has reached zero, then
        BEQ     FREE            ; the note played is finished, otherwise
        DEC     DELAY           ; it decrements the delay,
        BRA     END             ; then we exit
FREE    LDA     NUMBER          ; X contains the number (*2) of the
                                ; note to play
        LDA     MUSIC,X         ; A contains the note and its duration.
        CMP     #$FFFF          ; If A = $FFFF then we are at the end
                                ; of the song
        BEQ     RESET           ; And you connect to RESET to restart
        AND     #$00FF          ; We keep only the note.
        ASL                     ; Multiply by 2 because the frequencies
        TAY                     ; are encoded over two bytes
                                ; then transfer to Y as an index.
        LDA     NOTE,Y          ; A now contains the frequency in 16 bits
        SEP     #$20            ; Set A to 8 bits. But the 8 bits of A
                                ; remain unchanged.
        STA     FREQLOW         ; Saving to memory frequency variable (low)
        XBA                     ; The 8 high-order bits of A are exchanged
                                ; with the low-order 8 bits.
        STA     FREQHI          ; Frequency backup (high).
        LDA     #$A0            ; Access to the control registers for
        CLC                     ; oscillator $00 or $01, depending on
        ADC     UNCOUP          ; who is playing the note
        STA     $C03E
```

```
        LDA     #$03        ; Shutdown this oscillator in case it
        STA     $C03D       ; hasn't stopped on it's own
        LDA     UNCOUP      ; Switch the UNCOUP variable
        EOR     #$01        ; to toggle the oscillator used
        STA     UNCOUP
        LDA     #$00        ; Program the oscillator frequency
        CLC
        ADC     UNCOUP
        STA     $C03E
        LDA     FREQLOW
        STA     $C03D
        LDA     #$20
        CLC
        ADC     UNCOUP
        STA     $C03E
        LDA     FREQHI
        STA     $C03D
        LDA     #$A0        ; Activate oscillator with "One-shot"
        CLC                 ; mode on the right channel
        ADC     UNCOUP
        STA     $C03E
        LDA     #$02
        STA     $C03D
        REP     #$30        ;
        LDA     MUSIC,X     ; The X-register which is always at 16 bit,
                            ; contains the value of the number of
```

```
                                ; the note.
            XBA                 ; A is switched to the duration
            AND     #$00FF      ; And we only keep the duration.
            STA     DELAY       ; Store duration in the DELAY variable
            INX                 ; Next note
            INX
            STX     NUMBER      ; We save in NUMBER
            BRA     END         ; And it's done
RESET       LDX     #$0000      ; Resetting the NUMBER variable
            STX     NUMBER
FIN         CLC                 ; CLC before exiting an interrupt routine
            RTL

*=======================================
* Data Tables and Variable Storage
*=======================================

; Conversion table note -> frequency (frequencies are encoded w/ 16 bits)
NOTE        HEX     60008000A000C000
            HEX     E000000120014001

NUMBER      HEX     0000        ; 2 bytes reserved to store the note
                                ; number.  Since the notes are 2 bytes,
                                ; the number is multiplied by 2.
UNCOUP      HEX     0000        ; The UNCOUP variable is used to determine
                                ; if oscillator $00 or $01 is to be used.
```

```
; Table containing the music number according to the following structure:
; One byte for the note, followed by one byte for the duration.
; A $FFFF pair means the end of the song
MUSIQUE     HEX      0110011002200330
            HEX      0110052005200340
            HEX      0610071006100510
            HEX      02200220FFFF
```

Certainly this example only plays music on one voice, while the IIGS has 32 oscillators. Taking into account that it is preferable to take two oscillators per instrument and that oscillator $1F is used for the tempo, this gives us 15 usable voices plus the oscillator $1E free.