# Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 8

Content

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

Phone: 49 18 00 77
Postgiro: 7 24 23 44

## AUDIO

In this letter, we will review Amiga's audio capabilities. What is sound anyway? Sound is a wave which propagates in air, just like waves in water. Sound waves have a different form, depending on the sonic character. A powerful sound has a high waveform (high amplitude). An almost inaudible sound has a low waveform (low amplitude). In between there are infinite number of variations.

An example of a waveform, you can see in Figure 1 at the end of this issue. The figure you see is an ordinary waveform – a so-called "sine" wave. Waves of this type are regularly heard as a stable tone. Other waveforms will result in other types of sounds.

Well, so far, it is simple enough, but ...

... how does a pc sound? It does in principle in two ways. The first method is that it uses a so-called waveform generator. This generator can produce some different waveforms. Commodore 64 uses such a generator, where you can choose between "sine", "triangle", "rectangular" and "noise" waves. With these basic forms you can get quite complex sounds (tones). One can imitate various instruments like guitar, organ, strings (e.g. violin), trumpet, etc. The more advanced the wave generator, the better it is able to imitate different instruments.

The second way to produce audio is by using the sample method. It is the approach AMIGA uses.

SAMPLE, what is it? A sampler is also a waveform, but it is not fixed like "sine waves" are. If we try to imitate a piano a pure "sine" wave should be a very poor match to the original. A piano string produces a variety of sine waves at once. These waves overlay each other in a certain way which is specific for this instrument.

Every instrument has its unique composition of more or less pure sine waves. A trumpet also requires a number of sine waves together to produce sound. But both instruments create in a completely different way. It is therefore the human ear can hear the difference.

Since the piano sound does not only consists of a single sine wave (a perfectly smooth and full tone), but has a bit special "mix" we use the sample method to better match the piano sound.

When using the sample method, we "record" the sound with the Amiga. This is done by using a so-called SAMPLER. A sampler can also be called an A/D-Converter an analog-digital converter because it converts an analog signal into a digital signal.

Let us start by explaining what digital values are. A digital value can only be 0 or 1 (just as in computers). An analog value (usually represented by a certain level of voltage) can, however, reach infinitely many values between, e.g., 0 and 1 - a voltage can course be of any value between 0 and 1 (if these are the outer limits), for example. 0.0001, 0.2, 0.314, etc.

All samples in the Amiga are 8-BITS SAMPLE. That means that the analog signal ("waveform") which enters the SAMPLER will be turn into an 8-bit value before the coming out again and then sent into the pc. As is known, a group of 8 bits is the same as a byte - which may contain values from 0 to 255 (or sign from -128 to 127).

Study FIGURE 2 at the end of this issue. This is an example of how a sine-wave is sampled. Notice that here we use a signed value to specify the sample data. The vertical lines indicate where the analog signal is transformed into a digital signal. In this way, the AMIGA directly reads from the analog signal (waveform), and stores it in memory (or disk) as a digital value, i.e. as 1 (one) or 0 (zero).

To get the Amiga to play back a sample (a sampled sound), the built-in D/A-converter is used. Figure 3 shows how the analog signal will look like when it played on Amiga. As you see the signal is not as smooth as it was before the conversion (FIGURE 2). We can improve this by reading out the analog signal more often (with the A/D converter). This results in that the vertical markings in the figure will be closer together (get more reading points). This gives us a more accurate representation of the sound/music is playing.

However, there is the disadvantage of sampling, it consumes much memory. If we sample an audio 10,000 times per second - which is an ordinary sampling rate (means speed) on Amiga, and the sound takes 5 seconds the digitized date will "consume" 50,000 bytes (or approx. 49 Kb). We could point to a general compact disk player (which also uses the sampling method using a 16 bit D/A-converter with a sampling rate of 44,100 times per second. If it were technically possible for Amiga, the 5 seconds of audio using 220,500 bytes (approx. 215 kb).

After this brief introduction, how sounds can be created with computers and how sampling works, we will proceed with the topic "How to use sampling in AMIGA".

## AUDIO II

In the AMIGA there are 4 audio channels. This means that there may be played up to 4 different sounds/or samples) simultaneously. The volume (and playback speed) can be adjusted independently for each channel.

Register setup for the audio portion:

**AUDxLCH/L registers:**
AUDIO-pointers in AMIGA are used to identify the beginning of the sampling data in memory. Note that one pointer actually consists of "two" registers (HIGH and LOW) in the same manner as such bitplane pointers.

| NAME | BITS | ADDRESS |
|---|---|---|
| AUD0LCH | 16-31 | $DFF0A0 |
| AUD0LCL | 0-15 | $DFF0A2 |
| AUD1LCH | 16-31 | $DFF0B0 |
| AUD1LCL | 0-15 | $DFF0B2 |
| AUD2LCH | 16-31 | $DFF0C0 |
| AUD2LCL | 0-15 | $DFF0C2 |
| AUD3LCH | 16-31 | $DFF0D0 |
| AUD3LCL | 0-15 | $DFF0D2 |

Audio Address Registers

AUDxLEN registers:
AUD means AUDIO, LEN means length and "x" indicates the audio channel (0, 1, 2 or 3) to be used. This register is used to tell the AMIGA, how long the sample data in memory is.

| NAME | ADDRESS |
|---|---|
| AUD0LEN | $DFF0A4 |
| AUD1LEN | $DFF0B4 |
| AUD2LEN | $DFF0C4 |
| AUD3LEN | $DFF0D4 |

AUDxLEN Registers

Register AUDxLEN may contain a 16-bit value (0-65535). The only thing you must remember is that the length is given in words. So: If you have a sample of 5000 bytes in length, you put 2500 into this directory (NOTE: 1 word = 2 bytes).

The AUDxPER registers:
Its pupose is to set the sampling period (period = time/speed), "x" stands for the channel 0, 1, 2 or 3). This register is used to give the AMIGA the information on the sample's "playback speed.

| NAME | ADDRESS |
|---|---|
| AUD0PER | $DFF0A6 |
| AUD1PER | $DFF0B6 |
| AUD2PER | $DFF0C6 |
| AUD3PER | $DFF0D6 |

AUDxPER Registers

This register contains the velocity which the sample must be played with and may hold values from 124 to 65,535, where 124 is highest speed and 65535 is the lowest speed. If you set the register to a value less than 124, the AMIGA will skip the some sample data, so the sound becomes wrong (although not always audible). We will explain more about this register later in this issue when we walk through the program examples.

The AUDxVOL registers:
Its pupose is to set the volume where "x" indicates the channel 0,1,2 or 3. The volume can be set between 0 and 63 with bit 0-5 where 0 is silence and 63 is full volume. If bit 6 of this register is set to "1" the values of bit 0-5 are ignored and the highest volume (volume) is used. Imagine BIT BIT 0-6 as a group and you can vary the volume from 0 to 64 (65 different steps). BIT 7-15 are not used.

| NAME | ADDRESS |
|---|---|
| AUD0VOL | $DFF0A8 |
| AUD1VOL | $DFF0B8 |
| AUD2VOL | $DFF0C8 |
| AUD3VOL | $DFF0D8 |

AUDxVOL Registers

In the following chapter we walk through the program examples MC0801 and MC0802 and explain these topics in more detail.

## AUDIO III

We now review the program example MC0801:

Line 1: This switches off the DMA for audio channel 0 (if it should be set). See also setup DMACON in issue III.

Line 3: Load the effective address of the "sample" into the A1.

Line 4: Moving the address of the "sample" (A1) into $DFF0A0

Line 5: Move the value 48452 into $DFF0A4 (AUD0LEN) which represents the length in words so the sample we should play is 48452 * 2 = 96,904 bytes long (approx. 95 kb).

Line 6: Sets the replay-speed to 700 (AUD0PER).

Line 7: Sets the volume (intensity) to 0 (AUD0VOL).

Line 9: Switch on the DMA for audio channel 0.

Program lines 11-22 gradually increase the volume to full strength and the replay-speed is also increased gradually from 700 to normal speed. The normal play speed is 180. This effect makes it sound like a turntable, which needs a little time until it reaches normal speed.

Program lines 28-37 do the opposite. The velocity and volume is lowered gradually, so it sounds as if you suddenly pull the plug while the turn table still plays.

Line 11: Moves 0 into D1. D1 is used as to increase the volume gradually.

Line 12: Moves 700 into D2. D2 is used here to increase the replay-speed gradually down to 180 (i.e., increase speed).

Line 13: Moves 64 into register D7 (this register is used as a loop-counter). The loop therefore is performed 65 times (REMEMBER: 0 to 64).

Line 16-17: Jumps to routine "wait". This is done 2 times. In this way we get a break of 1 / 25 seconds (2/50 seconds).

Line 18: Moves the value of D1 into $DFF0A8 (AUD0VOL).

Line 19: Moves the value in D2 into $DFF0A6 (AUD0PER).

Line 20: Adds 1 to the value in register D1 to increase the volume.

Line 21: Subtract 8 from the value in D2 to increase replay-speed.

| | |
|---|---|
| Line 22: | Subtracts 1 from D0. If D0 is larger -1, jump up again to the "up" label. |
| Line 25-26: | Waiting until left mouse button is pressed. |
| Line 28: | The value 64 is moved to D7 so that we are ready for a new loop. |
| Line 31-32: | Jumps to twice toe the subroutine "wait to get a break of 1/25 second. |
| Line 33: | Moves the value of D1 into $DFF0A8 (AUD0VOL). |
| Line 34: | Moves the value of D2 into $DFF0A6 (AUD0PER). |
| Line 35: | Subtracts 1 from D1 (volume-count) to gradually decrease volume. |
| Line 36: | Adds 8 to D2 (speed counter) to gradually decrease replay-speed. |
| Line 37: | Subtracts 1 from D0. If D0 is larger than -1 jump up again to the "down" label. |
| Line 39: | Turns off DMA for audio channel 0. |
| Line 41: | Ends the program. |
| Line 43-55: | These should be known by now, but you might wonder why we use both line 200 and line 201? This is because the routine is called twice in succession. Imagine that we had removed the program, lines 50-55. First walk through would go well. But the second time just immediately after the electron gun will stop draw line 200, so that the routine will be completed premature. |
| Line 60: | Here we reserved space for the sample-data which is on the course disk. 1. |

Before the program can be run, it must be assembled, then read file "SAMPLE" in which is in the directory "issue 08" on the course-disk and finally start program with "j".

```
Seka> a
OPTIONS
No errors
Seka> ri
FILENAME> brev08/sample
BEGIN> Samplers
END>
Seka> j
```

Tip: To run the program several times without having it to assemble, put a label into the first line in the program. It may, for example. look like:

```
1 start:
```

Then you write this every time you want to start the program:

```
Seka> j start
```
"j start" means, jump to the label "start" and run the program from there.

We hope this explanation helps you to understand what happens in the machine when a sample is replayed. Do not forget to experiment with different values in the program to find out new effects and insights. It is very instructive!

And now we continue with the program example MC0802:

| | |
|---|---|
| Line 1: | Turns off DMA for audio channel 0. |
| Line 3: | Loads the effective address of the "sample" into A1. |
| Line 4: | Moves the address in A1 into $DFF0A0 (pointer to audio channel 0). |
| Line 5: | Sets the length of the sample to 8 WORD (16 bytes). |
| Line 6: | Moves 0 into $DFF0A6 (AUD0PER). |
| Line 7: | Moves 0 into $DFF0A8 (AUD0VOL). |
| Line 9: | Turns on DMA for audio channel 0. |
| Line 11: | Loads the effective address of "music" into the A1. |
| Line 13: | Here begins the routine, which plays different samples. |
| Line 14: | Branches to routine "wait". This routine will create a break of 5/50 seconds (a tenth of a second). |
| Line 16: | Moves the value, the address in A1 points to, into register D1, then increase the address in A1 by 2 (Remember: MOVE.W => + increases by 2 bytes). |
| Line 17: | Moves the value of D1 into $DFF0A6 (AUD0PER). |
| Line 18: | Move the next value in the "music" table to D2. |

Line 19: Moves the value in D2 into $DFF0A8 (AUDOVOL).

Line 21: Compares D1 with 0.

Line 22: If D1 is not 0, jump back to the "main loop" label.

Line 23 Compares D2 0.

Line 24: If D2 is not 0, jump back to the "main loop" label
So: If both D1 and D2 are equal to 0, the program will continue until the 26th line.

Line 26: Turns off DMA for audio channel 0.

Line 27: Ends the program.

Line 29: Here begins the routine that creates a pause of 5/50 seconds (a 10th of a second).

Line 30: Move the constant value 4 quickly to D1. D1 is used as a counter (the "waiting" 1/50 seconds 5 times).

Line 33-37: Waiting until the electronic beam has reached line 200.

Line 40-44: Waiting until the electron beam has reached screen line 201. The reason why we should expect both display lines 200 and 201 are explained in the review of example MC0801.

Line 46: Subtracts 1 from the value in Dl. Check if Dl is -1, if not, branch back again to the "wait2" label. So: Program lines 33-44 are performed 5 times (5 * 1/50 seconds delay).

Line 48: Branches back to the calling instance. Here return to the program line 14 and continues from there.

Line 51: Here is the sample which is replayed. These data represent a sine wave. Notice that the DMA automatically replays the sample data on and on again so that there will be an infinitely long sine-wave.

Line 54-83: Here is the data which controls the tones to be played.

The first value must be loaded into AUD0PER register. The second value is introduced in AUD0VOL register. The third value is introduced in AUDOPER register, and so on until all data (tones) are played.

What happens when the program runs is that the first two values will be entered into AUD0PER and AUD0VOL respectively and then a delay of a 10th of a second will take place. After this delay the next two values are put into AUD0PER and AUD0VOL.and another delay of a 10th of a second takes place. This continues until 0 it is loaded both into AUDOPER and AUDOVOL, indicating that the table is finished, and the program ends.

So the program lines 54-55 will play the tone "C" in 2 tenths of a seconds, then will come to halt (volume set to 0) at a tenth of a second before the next note is played.

At the back of this issue you find a table containing the values of the different tones in the scale. Try to put your own tones as you like.

In this chapter we have reviewed how to play a sample ready-made, and then how to make a single sound program that can play different tones. Now it's your turn. Remember Practice makes perfect.

## MACHINE CODE VII

In this machine code section we review the instructions “mulu”, “muls”, “divu” and “divs”.

Let us begin by mulu, which stands for: MULtiply Unsigned. Mulu performs a multiplication with Unsigned values.

```
        MOVEQ       #10, D1
        MULU        #5, D1
```

First line moves the value of the constant 10 quickly into D1. The second line multiplies D1 by 5, so that D1 will contain the 50. The instruction “mulu” multiplies two 16bit words and result in a 32bit long-word.

The next instruction we need to look at is “muls” it stands for (which you probably already have guessed) MULtiply Signed. The difference between “mulu” and “muls” is that “muls” multiplies two signed words and results in a signed long word as a result.

Example:
```
        MOVEQ       #10, D1
        MULS        #-5, D1
```

D1 will now contain -50 (FFFFFFCE).

The divu-instruction we examine now stands for: DIVision Unsigned. This instruction takes A 32bit long word and divides it with a 16bit Word. At the result we will take a closer look.

But first let us take an example:

```
        MOVEQ       #500, D1
        DIVU        #10, D1
```

and one for ......

```
        MOVEQ       #10, D1
        DIVU        #3, D1
```

and yet another ......
```
        MOVE.W      #1001, D1
        DIVU        #2, D1
```

The first example will give the following value in D1: $ 00000032 (or 50 decimal).

The second example is the number 10 which is divided by 3 => the result is equal to 3.33. It looks a bit "messy" out, right? Given that the binary numeral system can only contain integer Dl will look like this: $000100003 BIT 0-15 ($0003) gives the integer, and BIT 16-31 ($0001) gives the rest. (If we had divided by 5, then BIT 16-31 contains 0 since there's no rest). So the result can be interpreted by: 3 plus a third, or 3.33.

The last example also has a rest. D1 will now look as follows: $000101F4. So: 500 ($01F4 = 500 decimal) plus 1 ($0001) half, which may be interpreted as written 500+1/2 or 500.5.

The last instruction "divs" works the same way as DIVU, but operates with signed numbers.

These instructions, although they are very effective, are not used very often. It is most when writing complex programs with a lot of mathematical calculations that they come to their full potential.

This was the machine code chapter of this issue. In the next issue we look at INTERRUPTS and related instructions.

## MIDI

This chapter explains a little about MIDI. The abbreviation MIDI stands for "Musical Instrument Digital Interface" which means digital interface between musical instruments.

Musical instruments equipped with MIDI are more prevalent with keyboards (electronic pianos or SYNTHESIZER). These instruments can be linked together via a MIDI interface. They may for example coupled so that when you play KEYBOARD one, the other would play exactly the same, but with a different kind of sound. One can say that via MIDI you can control many keyboards (or SYNTHESIZER) from one master keyboard.

Going one step further, you can connect a keyboard with a PC. This opens up a myriad of opportunities. The most common use of this connection is to "record" first the melody and then "play" the bass (as a Playback) while the next voice is played. In this way you can build full featured song even an entire symphony, step by step.

The actual MIDI transmission is divided into channels (channels). If for instance two keyboards are coupled with a PC you have the possibility to use the keyboards in a way that they play totally different things. You allow keyboards only to "listen" to various channels. A MIDI interface has 16 such channels.

Here are some technical explanations on MIDI. MIDI uses care of a serial transfer method. Serial means that transmitted data is sent or received one bit at a time. To get a byte you must then send the 8 bits in succession. The upload speed of MIDI is 31,250 bits per. seconds (denoted BAUD). So that can be transmitted up to 31250/8 = 3906 Bytes per second (in practice there will be a small break between each eighth BIT transfer so that we can expect approx. 3000 bytes (approx. 3kB) per seconds as the maximum transfer rate.

This transfer rate is more than enough if you look at what is transmitted over MIDI. When you press a key down, it transfers, only 3 bytes. The first byte tells what channel is sent in, and what type of information is transferred (in this case "key down" or "NOTE ON", as it says on the MIDI language). The second byte contains which key was pressed while the third byte contains, how hard it was pressed (known velocity).

You've got a simple introduction to what MIDI is and how this works in its simplicity. In issue XII are some simple MIDI routines for the Amiga. To use these routines you must have a keyboard (which has MIDI) - and a MIDI interface)

## SOLUTIONS FOR TASKS IN ISSUE VII

TASK 0702:

| BINARY | UNSIGNED | SIGNED |
|---|---|---|
| 01101001 | 105 | 105 |
| 10111110 | 190 | -66 |
| 11110001 | 241 | -15 |
| 00101101 | 45 | 45 |
| 11010011 | 211 | -45 |

TASK 0703: The lowest value with a bit-group of 20 can contain is -524388 (decimal) and the highest value is 524287 (decimal).

TASK 0704: This program can, for example look like this:

```
NOT.W     D1
ADD.W     #1, D1
RTS
```

## TASKS FOR ISSUE VIII

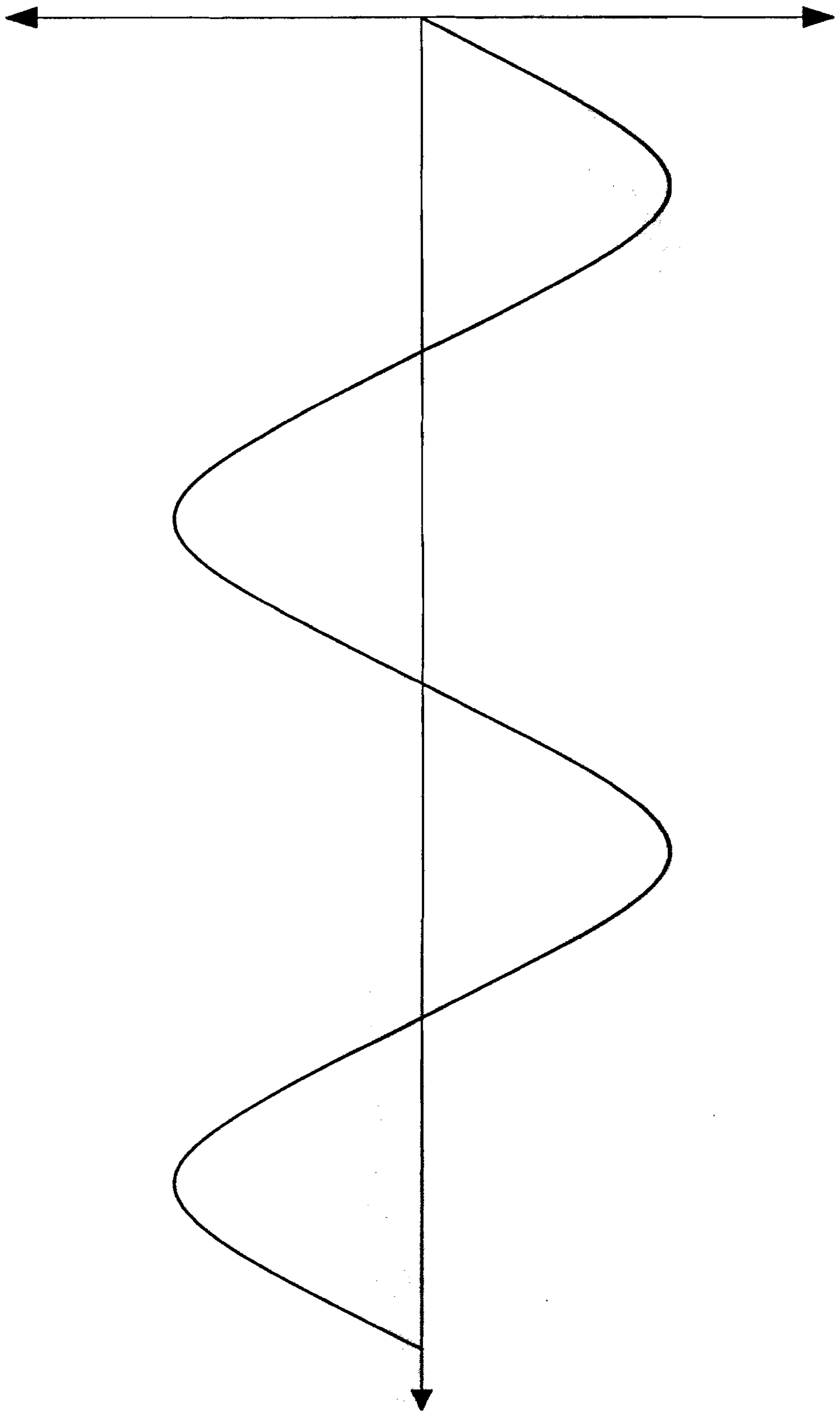
TASKS 0801: What does the Amplitude of the sound?

TASKS 0802: What is the method of sampling?
TASKS 0803: What is an A/D-converter?
TASKS 0804: What will be the result in D1 after execution of this program (try without K-Seka!):

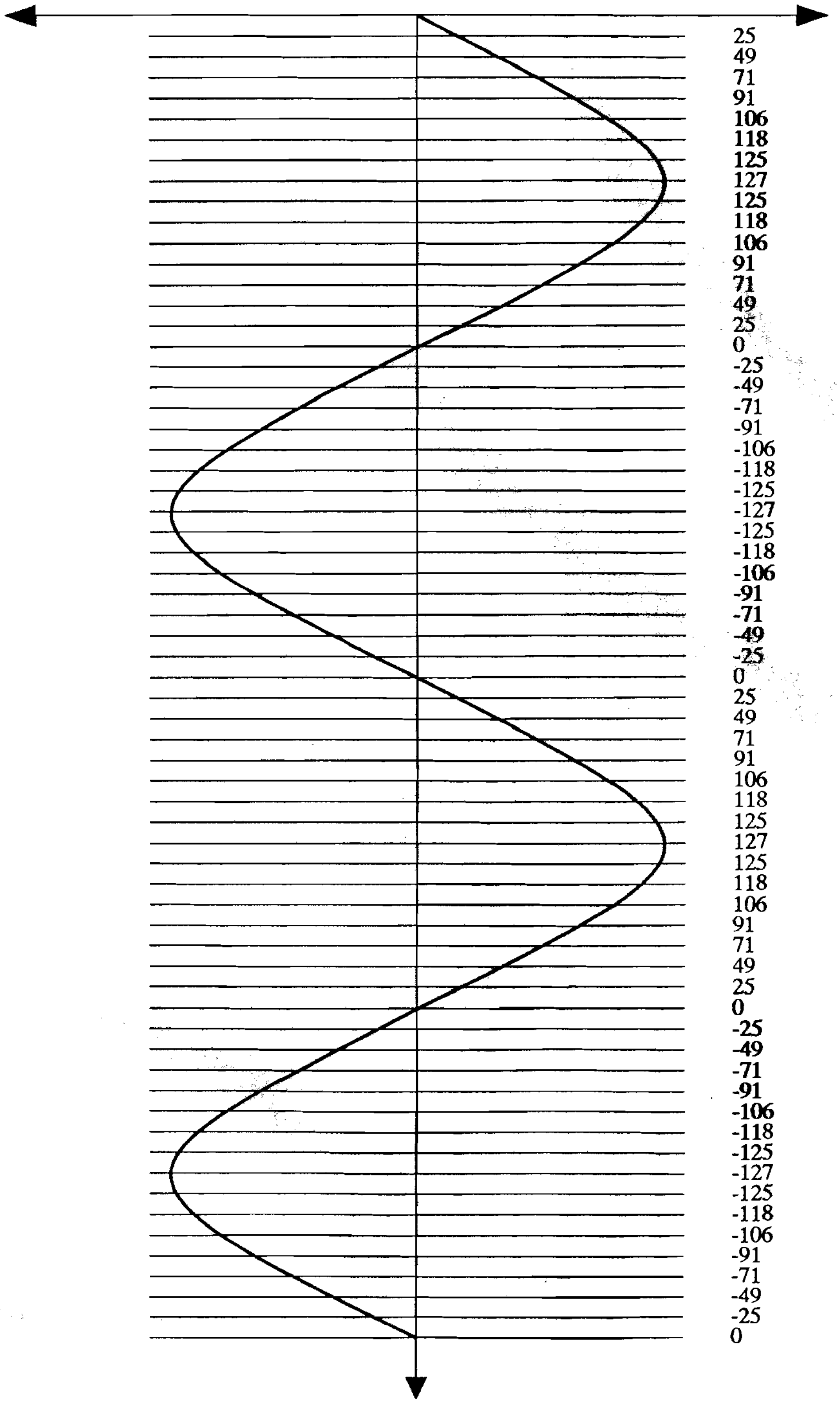
```
MOVE.W    #-75, D1
MULS      #-3, D1
RTS
```

TABLE OF VARIOUS TONES and SAMPLING RATE

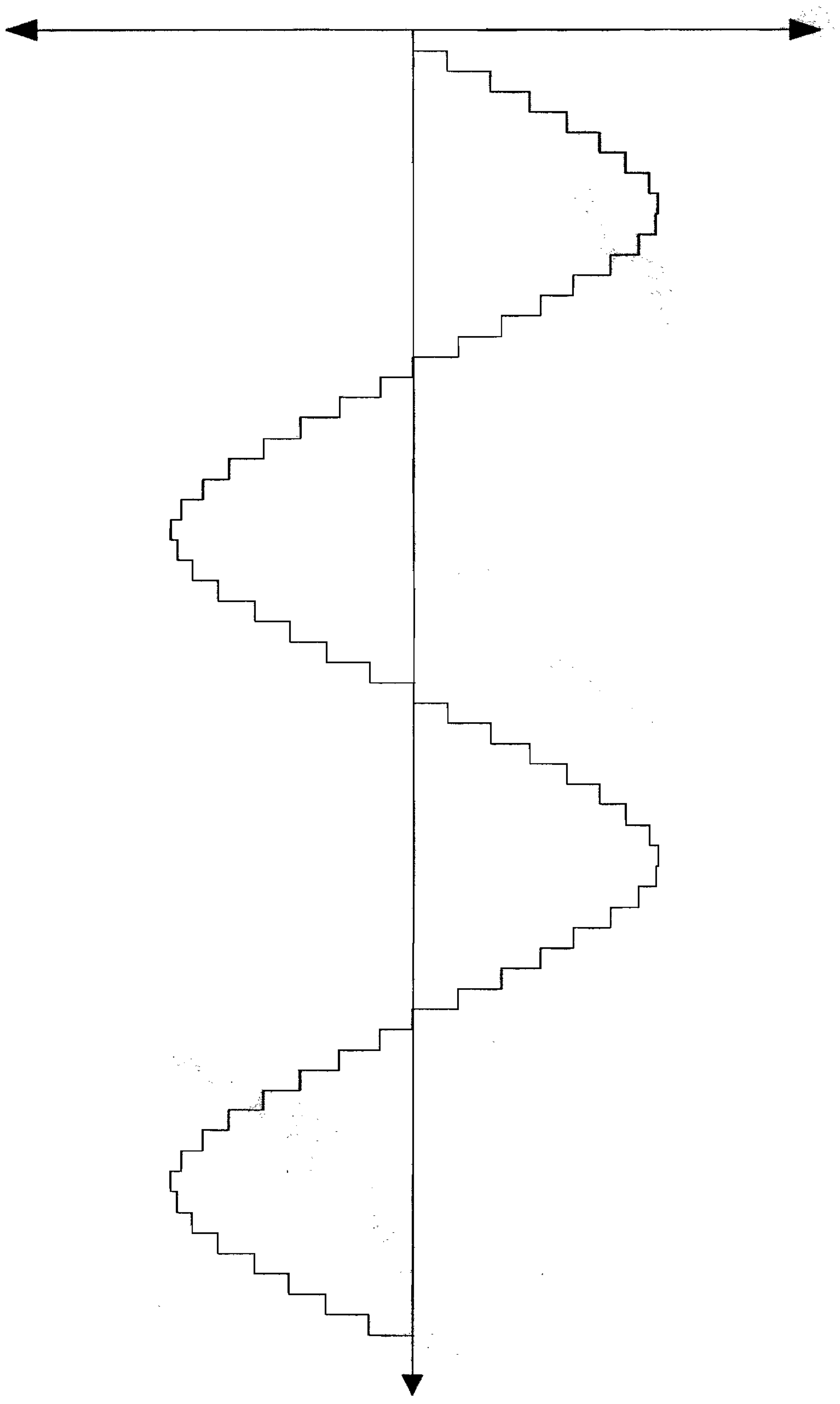
| TONE | PERIOD (RATE) |
|---|---|
| A1 | 508 |
| A1# | 480 |
| H1 | 453 |
| C2 | 428 |
| C2# | 404 |
| D2 | 381 |
| D2# | 360 |
| E2 | 339 |
| F2 | 320 |
| F2# | 302 |
| G2 | 285 |
| G2# | 269 |
| A2 | 254 |
| A2# | 240 |
| H2 | 226 |
| C3 | 214 |
| C3# | 202 |
| D3 | 190 |
| D3# | 180 |
| E3 | 170 |
| F3 | 160 |
| F3# | 151 |
| G3 | 143 |
| G3# | 135 |
| A3 | 127 |



Figur 1.


25
49
71
91
106
118
125
127
125
118
106
91
71
49
25
0
-25
-49
-71
-91
-106
-118
-125
-127
-125
-118
-106
-91
-71
-49
-25
0
25
49
71
91
106
118
125
127
125
118
106
91
71
49
25
0
-25
-49
-71
-91
-106
-118
-125
-127
-125
-118
-106
-91
-71
-49
-25
0

Figur 2.



Figur 3.