# NGPC Sound Tutorial

Ivan Mackintosh, 15 Sept 2001

## *Introduction*

This tutorial only covers using the synthesised sound capabilities of the NGPC and does not cover playing sound samples.

The first thing you need to be aware of is that the NGPC has 2 processors. The TLCS900H (main) processor and a dedicated Z80 processor that has access to the sound hardware. The memory maps for these processors are completely separate apart from one 4kb area of RAM. This means that the main processor cannot directly access the sound hardware and conversely the Z80 processor cannot access any of the other bits of NGPC hardware (screen etc. ).

Because of this relationship between the 2 processors a 'sound driver' (piece of code residing in the Z80 memory map) has to be written in order for your NGPC game to be able make any sound. This document describes how to create a sound driver and the techniques used for communication between the 2 processors.

The great news for NGPC programmers is that the most common assembler used by homebrew coders: "AS", can also be used to assemble the Z80 sound driver code.
In the same way that you would specify the TLCS900 processor at the top of your NGPC program:

```
CPU 93C141
```

You can specify you want to assemble a Z80 program at the top of your code as follows:

```
CPU Z80
```

Unfortunately, you cannot mix and match your languages in a single ASM file.

## *Sound Hardware Registers*

The NGPC can play 3 tones and 1 noise channel simultaneously. These are controlled by 2 registers:

| Z80 Memory Address | Register Name | Register Description |
|---|---|---|
| 4000h | REG_CH4 | Controls the noise |
| 4001h | REG_CH123 | Controls tones 1 to 3 |

These are the only sound registers. They can be used to send control information as well as data to the sound hardware.

These registers are only 8bits and therefore some operations require 2 writes to the register. The topmost bit (bit 7) is set to 1 to indicate the first byte of an instruction to the sound hardware.

The next 3 bits (bits 4, 5 and 6) specify the operation. These are listed in the table below:

| Value | Register Name | Description | Byte transfers required |
|---|---|---|---|
| 000 | TONE1 _FREQ | Set the frequency for tone channel 1 | 2 |
| 001 | TONE1 _VOL | Set the volume for channel 1 | 1 |
| 010 | TONE2 _FREQ | Set the frequency for tone channel 2 | 2 |
| 011 | TONE2 _VOL | Set the volume for channel 2 | 1 |
| 100 | TONE3 _FREQ | Set the frequency for tone channel 3 | 2 |
| 101 | TONE3 _VOL | Set the volume for channel 3 | 1 |
| 110 | NOISE_ CTRL | Set the type of noise required | 1 |
| 111 | NOISE_ VOL | Set the volume for the noise channel | 1 |

The remainder of the bits (0 to 3) are used for data transfer. In the case of the volume control this provides enough space for the entire instruction to be carried out in a single data transfer.

For frequencies a further write has to be carried out to send the rest of the data to the sound hardware.

## *Setting the Volume*

I will start with the volume control as it is the simplest to understand.

| Bit number | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Value | 1 | Op | Op | Op | V | V | V | V |

You will notice that bit 7 is always set to “1” to mark the first transfer for this operation.
The values for “Op” should be one of the volume registers taken from the previous table.
The values for “V” is the volume itself and range from 0 (loudest) to 15 (silence) .

```
LD      IX,     4001 H
LD      ( IX) ,  10111101  B
```

So, if you have been keeping up you should be able to decipher the code above to mean set the volume of channel 2 to be 13 (a fairly quiet volume) .

This would be a little more self explanatory if equates had been previously defined:

```
REG_CH123       EQU     4001 H

FIRST_ TXFER    EQU     10000000  B
TONE2_VOI       EQU     00110000  B
```

The same code could then be written in a more readable fashion:

```
LD      IX,     REG_CH123
LD      A,      FIRST_ TXFER
OR      A,      TONE2_VOL
OR      A,      13              ; set the volume to 13
LD      ( IX) , A
```

If like me, you tend to think of a volume of 15 being the loudest and 0 the quietest then here’ s the same code again but with the addition of a volume inversion.

```
LD      IX,     REG_CH123
LD      B,      13              ; set the volume to 13
LD      A,      15
SUB     A,      B
OR      A,      FIRST_ TXFER
OR      A,      TONE2_VOL
LD      ( IX) , A
```

There are a couple of things to note here:
Regarding the volume, our value of 13 is now inverted (15 minus 13 = 2 ) and therefore 13 is now a loud value as opposed to a quite one.
The other thing that caught me out (due to having no previous Z80 programming experience) is unlike the TLCS900 whereby any register can be used for anything, in Z80 land all computations go through the accumulator ('A') register. Therefore the inverting has to be done first due to the SUB instruction leaving the result in 'A' and then the logical OR instructions have to be applied afterwards, the result of their effect also being stored in 'A'. Fortunately, the assembler will catch any mistakes of this nature.

## *Setting the Frequency*

The frequency can be in the range of 0 (high pitched tone) to 1023 (low pitched tone) making a total of 10 bits of data that need to be transferred to the sound hardware. This will require 2 writes.

The format of these writes is as follows:

| **Bit number** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Value** | 1 | Op | Op | Op | F3 | F2 | F1 | F0 |

| **Bit number** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Value** | 0 | | F9 | F8 | F7 | F6 | F5 | F4 |

As with setting the volume, bit 7 signifies if it is the first transfer (Note that it is set to 0 for the second. Bits 4 to 6 specify the channel to set the frequency of – see the previously table for the value of these. And the 10 bit frequency is split over the 2 bytes.
Note that bit 6 on the second byte is unused.

```
LD      IX,     REG_CH123
LD      A,      FIRST_ TXFER
OR      A,      TONE1_FREQ
OR      A,      0DH
LD      ( IX) , A               ; write first byte
LD      A,      2DH
LD      ( IX) , A               ; write second byte
```

The above code sets the frequency of the tone channel 1 to the hex value of 2DD. Note that the first transfer flag (bit 7) is only set for the first transfer.

The NGPC frequency setting of 2DD refers to the music note C-3. The following table provides the frequency settings for rest of the music notes:

| | Octave | | | | |
|---|---|---|---|---|---|
| Note | 2 | 3 | 4 | 5 | 6 |
| C | | 2DD | 16E | 0B7 | 05B |

| | | | | | |
|---|---|---|---|---|---|
| C# | | 2B4 | 159 | 0AD | 056 |
| D | | 28D | 146 | 0A3 | 051 |
| D# | | 269 | 134 | 09A | 04D |
| E | | 246 | 123 | 091 | 048 |
| F | | 225 | 112 | 089 | 044 |
| F# | | 206 | 103 | 081 | 040 |
| G | | 1E9 | 0F4 | 07A | 03D |
| G# | | 1CE | 0E7 | 073 | 039 |
| A | 368 | 1B4 | 0DA | 06D | 036 |
| A# | 337 | 19B | 0CD | 066 | 033 |
| B | 309 | 184 | 0C2 | 061 | 030 |

## *The Noise Channel*

The sound register REG_CH4 (memory address 4000h) handles the noise channel. A single byte transfer is all that is required to control the noise channel.

The format of the noise control byte is as follows:

| **Bit number** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **Value** | 1 | Op | Op | Op | | FB | N1 | N0 |

As with the volume and frequency settings, bit 7 is set to 1 to signify first data write, bits 4 to 6 represent the operation.

Bit 3 remains unused.
Bit 2 represents feedback.
Bits 0 and 1 operate in the following way: the binary values 00, 01, 10 represent fixed noise tones at 3 different pitches. The value 11 instructs the sound hardware to use the frequency as set by tone channel 3. Note that this channel 3 is separate to the tone channel 3 we have mentioned previously as it applies to the REG_CH4 register and does not affect the REG_CH1 23 register in anyway.

The frequency as volume can be set up exactly the same way as with the REG_CH1 23 register, described in previous sections.

```
; set up the desired tone on for tone 3 of the REG_CH4 register
LD      IX,     REG_CH4
LD      A,      FIRST_TXFER
OR      A,      TONE3_FREQ
OR      A,      05H
LD      (IX),   A
LD      A,      03H
LD      (IX),   A

; apply the required volume
LD      A,      FIRST_TXFER
OR      A,      NOISE_VOL
OR      A,      0       ; loudest
LD      (IX),   A

; apply the noise setting
LD      A,      FIRST_TXFER
OR      A,      NOISE_CTRL
OR      A,      3       ; N0 and N1 = 1 (use frequency from tone 3)
```

## *The Simplest Z80 Driver Example*

The following bit of code is the smallest amount of Z80 code required to get your NGPC to make a noise. In this case it is play the tone C-3 on channel 1 at the loudest volume. You should be able to assemble code as-is. Note that there is no hardware ability to set a duration and therefore this C-3 tone will play indefinitely.

```
        CPU     Z80
        ORG     0

REG_CH123       EQU     4001 H

FIRST_ TXFER EQU        10000000 B
TONE1_FREQ      EQU     00000000 B
TONE1_VOL       EQU     00010000 B

        LD      IX,     REG_CH123

        ; set the volume
        LD      A,      0               ; loudest volume
        OR      A,      FIRST_ TXFER
        OR      A,      TONE1_VOL
        LD      ( IX) , A

        ; set the tone ( C-3' s tone frequency is 2DDH)
        LD      A,      0DH             ; bits 0-3 of tone C-3
        OR      A,      FIRST_ TXFER
        OR      A,      TONE1_FREQ
        LD      ( IX) , A
        LD      A,      2DH             ; bits 4-9 of tone C-3
        LD      ( IX) , A

END:    JP      END                     ; stop the Z80 executing random memory
```

## *Installing the Z80 Driver*

As mentioned previously the there is 4kb of shared RAM between the 2 processors. The address of this RAM in the TLCS memory map is 7000 h and in the Z80 memory map it is 0. When a Z80 processor is initialised its program counter is zero and thus you must code your driver so that there is code at this address (using ORG 0).

The Z80 processor can be disabled and reset via the TLCS register at 0B8H. You are probably one step ahead of me now and thinking that it is a simple case of disable the Z80, copy the driver code to Z80 RAM at address zero and then re-enable the Z80 again. The driver will then be executed. If you thought that then you are absolutely correct

Here' s a TLCS code routine to do just this:

```
InstallSoundDriver:
        LDW     (00 B8H),       0AAAAh          ; stop the z80 and sound chip

        ; copy the driver into memory
        LD      BC,             SoundDriverEnd - SoundDriver
        LD      XDE,            7000 h          ; where the z80 shared memory is
        LD      XHL,            SoundDriver
        LDIR    ( XDE+),        ( XHL+)

        LDW     (00 B8H),       05555 h         ; start the z80 and sound chip
        RET
```

```
SoundDriver:
        binclude    "mydriver.z80"
SoundDriverEnd:
```

All you need to do is assemble your driver code, slot in the appropriate filename in the binclude above and then call this routine from your TLCS code during initialisation:

```
        CALL    InstallSoundDriver
```

## *Communicating with the Z80 Driver*

OK, so in the last 2 sections we have created a simple sound driver, copied to the Z80 RAM and run it. Its not a very good sound driver as it only plays a single frequency indefinitely.

There are 2 main reasons why you would want to communicate with your sound driver once it is running:

- A more complicated sound driver (e.g. SFX) needs control information sent to it such as play sound effect number 1.
- The Z80 memory map does not include a clock of any sort. Therefore a pulse at regular intervals from the TLCS processor is required such that you sound driver can apply a duration to a tone.

The way this is achieved is by the Z80 polling specified shared RAM locations for the control information and clock pulses. The TLCS can then set these values as required.

For the clock pulse a simple:

```
        INC     (MyClockPulse)
```

can be added into the VB interrupt of the TLCS. MyClockPulse would be a memory location shared by both the TLCS and Z80.

Here's some Z80 code that polls for a clock pulse and control byte:

```
        CPU     Z80
        ORG     0

        LD      SP, 0FF0H       ; set the initial value of the stack
        JP      START           ; jump to the start of the driver

        ORG     10H             ; a fixed place in share memory for
                                ; communication

VBPULSE:    DB      0
CONTROL:    DB      0

        ORG     20H             ; the driver starts here
START:

        LD      A,      (VBPULSE)
        CP      A,      0
        JR      Z,      START
        LD      A,      0
        LD      (VBPULSE) , A

        LD      A,      (CONTROL)
```

```
CP      A,      CTRL_DO_SOMETHIN(  ;  your control command
JP      Z,      DO_SOMETHIN(       ;  jump to your routine
JR      START                      ;  continue waiting
```

Points to note from the above code:

- The communication values are defined early on in the code. This coupled with the ORG statements means that they have fixed values and if you change your sound driver your communication values do not move around in memory.
- The communications values are at location 10 H in the Z8 0 memory map. This equates to memory location 7010 H in the TLCS memory map. Therefore, in this case, the MyClockPulse value pulsed from the TLCS VB interrupt should equate to 7010 H.
- The program loops continuously until the VBPULSE is set once set. Once this is detected and the tight loop is exited the VBPULSE variable must be cleared to avoid the loop being instantly exited next time it is entered. Remember to do this for your control information variable too when you execute that code.

And that' s it!

If you would like to see how a more complicated driver fits together using the techniques mentioned in this document then you can download the source code for my sound effects driver from my web site:

http: //www. severnroad. co. uk/ ngp