Moving Sound from FLASH to RAM

Preferences

Sound_Generator.asm has to be as fast as possible to reduce the footprint of 'sound generation'. The standard MObject generates A4 with 440Hz with a 256 sample resolution. The results in

```
440/s * 256 samples = 112.640 samples/s
```

composition frequency. Assumed a 16MHz micro controller frequency this will give us

```
(16.000.000 cycles/s) / (112640 samples/s) = 142 cycles/sample
```

In short, whatever MObject does to produce sound, it has to be processed in **142** machine cycles.

Therefore the code producing the sound has to be as short as possible to enable as many 'parallel' tasks as possible – we don' t know what MObject will have to do in any specific application.

Given the fact, that Atmega micro controllers don't support interrupt hierarchies, sound production in MObject will be interrupt driven. All the rest can't!

Basic Sound Production

Sound production is a simple process. A sound is stored as array of sequential samples. These samples have to be read from memory and sent to an 8bit output port:

```
        MOV  address, start_of_sound
loop:
        LPM  sample, address
        OUT  sample
        INC  address
        JMP  loop
```

Processing Overhead

Unfortunately incrementing '`address`' does not do the trick alone. At the last sample of the sound '`address`' has to be reset to the '`start of sound `' address. Thus we have to:

1. Initialize a second address storage to the '`end of sound address`'
2. After incrementing '`address`' behind the last sound sample value, reset '`address`' to '`start of sound address`'

Which is sad! Because of addresses being 16bit values we need:

1. Two additional accumulators/registers for '`end of sound address`'
2. Two additional accumulators/registers for '`start of sound address`'
3. Cycles to compare '`address`' with '`end of sound address`'
4. Cycles to reset '`address`' to '`start of sound address`'

Like this (the green part of the code is what we wish to do!):

```
setup:
       LDI  start_of_sound_l, low (sound)
       LDI  start_of_sound_h, high(sound)
       MOV  end_of_sound_l, start_of_sound_l
       MOV  end_of_sound_h, start_of_sound_h
       ADD  end_of_sound_l, sound_len_l
       ADC  end_of_sound_h, sound_len_h
start:
       MOV  address_l, start_of_sound_l
       MOV  address_h, start_of_sound_h
loop:
       LPM  sample, address
       OUT  sample
       ADD  address_l, 1
       ADC  address_h
       CPSE address_l,  end_of_sound_l
       CPSE address_h,  end_of_sound_h
       JMP  loop
       JMP  start
```

which means:

1. Load 2 registers with start address to store it
2. Load 2 registers with address behind end address to store it
3. Load 2 registers with start address to run with
4. Load 1 register with sample value
5. Output the sample value
6. Add 1 to 2 registers of the run register
7. Compare 1 register of the run register with one register of the end register
8. Compare 1 (the other) register of the run register with one register of the end register
9. Jump to loop or
10. Jump to loop initialisation

Lazy hackers may believe this overhead does no matter as long as it works below the time limit it should work. Strict ones on the other hand may calculate that this would mean that most of the calculation power of the micro controller will be eaten away – one million times per 10 seconds – to nothing else but address calculation, address comparison and address initialisation. Not to think about the use of seven registers and the complexity of adding further functionality to the code by merging in additional code modules. This hurts!

Remembering that we have to fight for each cycle, in regard to further applications, one may think about other ways to do sound with less cycle consumption.

Shortcuts

- What if we only need three registers to do it all?
- What if we need to initialise only one address and only ones?
- What if we don't need to reset address pointers to initial values?

We start with a simple idea. AVR micro controllers present special 16 bit address pointers to the assembler in two ways: as 16bit value (X, Y, Z) and as pairs of 8bit values (XH, XL, YH, …). For example X is the same as XH:XL. Incrementing 8bit values let them grow from 0 to 255 (0xFF). Incrementing an 8bit register with content 0xFF will leave us with 0x00 *and* the carry flag set. This would make an address range from, for example, XH:0x00 to XH:0xFF where XH keeps whatever it is after initialisation.

This gives us a range of exactly 256 samples to enumerate through again and again without even thinking about the upper 8bit of the 16bit address.

So if we are prepared to life with sounds no longer (and no shorter!) than 256 samples in a row, we are almost done:

```
        MOV XL, low (start_of_sound)
        MOV XH, high(start_of_sound)
loop:
        LPM sample, X
        OUT sample
        INC XL
        JMP loop
```

That's it (not really but near it)!

Refinement

There are two points left to work on:

1. LPM needs 3 cycles, whereas LD only needs 1
2. If we start or stop a sound on a randomly chosen point, we may produce an acoustic click

Reduce machine cycles

To change LPM to LD the sound has to stay in RAM, not in FLASH. Not all AVR micro controller support reading program memory using an LD instruction, most do not.

Spending some RAM to the sound wave, we even may enhance the potential functionality of the program! If our program reads sounds persistent from the same location, we only need to move other sound waves to this location to change the sound without fiddling with complicated address calculation inside the sound play routine again.

So we simply move our sound wave from program memory to RAM like this:

```
        MOV XL, low (start_of_sound_ram)
        MOV XH, high(start_of_sound_ram)
        MOV ZL, low (start_of_sound_flash)
        MOV ZH, high(start_of_sound_flash)
copy:
        LPM sample, Z+
        ST  X+, sample
        JMP copy
```

But stop! Here we run into a problem. The theoretical probability of our RAM address to start with XH:0x00 is 1/256. In reality it is zero! But if it does not, our 'rotating through XL' put us into real trouble.

Here we have to pay some bytes of RAM.

If we start on a random position with our sound wave and the address contains other than 0x00 in it's lower byte, than we have to move further to find the next address fulfilling the requirement. This address will start at the next XH position. For example:

Natural address: `0x34:0x81 => 0x35:0x00`

In respect to the necessary storage, we than need 256 bytes with and behind this '*idealised*' address. If we preserve 256 bytes of RAM we may get an address range of:

```
0x34:81 to 0x35:0x80
```

Moving the start address of the block to let XL become 0x00 would mean:

```
0x34:81 to 0x35:0x80 => 0x35:0x00 to 0x35:0xFF
```

But all bytes behind `0x35:0x80` are not allocated! Which means, we have to double the RAM reservation to ensure our sound starts with XL=0x00 and ends inside the preserved memory range. We have to preserve 512 bytes for a 256 byte sound wave! With this, the code for the copy operation looks like this:

```
        MOV  XL, low (start_of_sound_ram)
        MOV  XH, high(start_of_sound_ram)
        MOV  ZL, low (start_of_sound_flash)
        MOV  ZH, high(start_of_sound_flash)
        CPSE XL, null
        INC  XH
        CLR  XL
copy:
        LPM  sample, Z+
        ST   X, sample
        INC  XL
```

```
        BRNE copy
```

Four lines of once-to-run assembler code and 256 bytes of RAM is all we have to pay. After the copy operation, thankfully, even the sound wave address registers are initialized!

We have to increment XL in an extra line for two reasons:

1. ST does not set any status flag
2. ST X+, … would increment XH which we do not wish do be done

Respecting the first case, we would have to use 'TST' to find out about XL becoming 0x00, which also would consume a cycle. Studying the assembly language reference of the AVR micro controller, the solution shown above is in general at least as compact as using ST with X+ followed by TST.

Prevent acoustic clicks

We have to do things to prevent clicks at start or end of sounds:

1. Sound waves have to start and end with output value 'NULL'
2. Sound wave reproduction has to start at sound wave sample array index 0
3. Sound output must not stop before sound address pointer is NULL