DMA and Gameboy (DMG) Sound

CS 2261: Media Device Architecture - Lecture 10

HANSEN – Feb. 16, 2022 – Georgia Institute of Technology

Overview

- I hope you've all *at least started* homework 3!
- DMA
- (Original) Gameboy Sound

Some Video Buffer Operations are *SLOW*

- What makes a program slow?

```
// super slow clear screen
for(int i=0; i<34800; i++)
{
  VIDEO_BUFFER[i] = 0;
}
```

Is there a solution?

- Our general purpose CPU is "slow."
  - 16.78MHz is pretty fast, but many operations take multiple clock cycles.
    - If setting a single pixel took ~3 cpu cycles, incrementing i took another 1, and checking for i being out of bounds took ~2 more, that clear screen would take ~38400*6 cpu cycles -- **roughly 230k cpu cycles**!
    - The VBlank period is only ~84k CPU cycles(TONC), so we've got a problem here!
    - Even VBlank + VDraw is only ~281k cycles, so filling the screen isn't going to work this way without flickering...
- Moving blocks of memory or filling blocks of memory is very common and fairly slow via the processor.

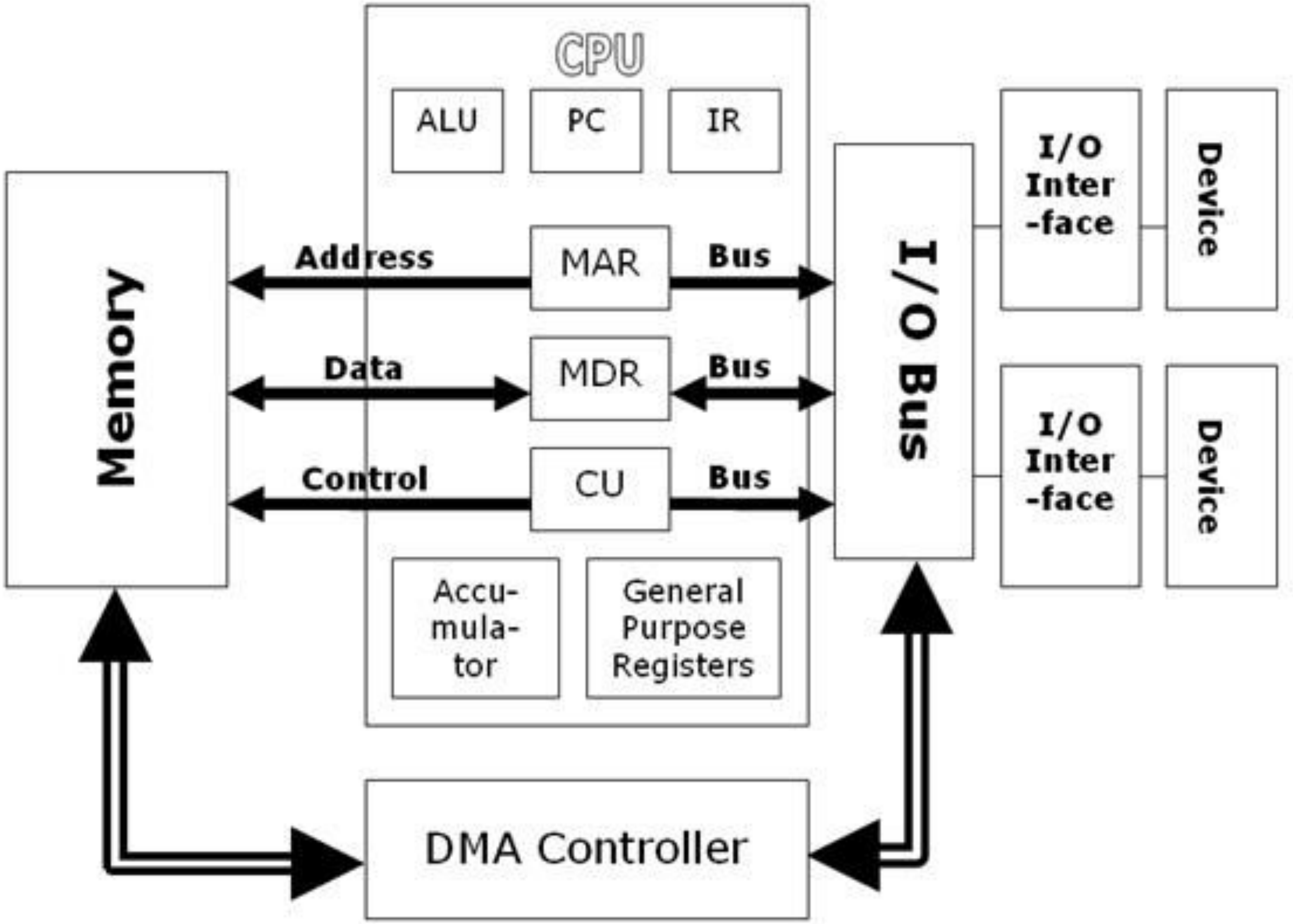
Custom Circuits

- Can do one thing (or maybe a few things) really quickly
  - (and often more efficiently from a power perspective)
- Used for common operations that could use the extra speed.
  - Often the operation is fairly simple.
- If things become too complicated, we need to use the general purpose processor.

Enter DMA

- DMA = ***<u>Direct Memory Access</u>***
- Dedicated hardware-backed data copy from one part of memory to another.
  - Up to 10x as fast as array copies
    - Especially for larger arrays
- You set it up, the CPU is halted, data is transferred, and CPU resumes where it left off.
- It's completely blind to the contents/type of the data being copied (so we'll use void pointers with it).

A Visual


CPU
ALU
PC
IR
Memory
Address
MAR
Bus
Data
MDR
Bus
Control
CU
Bus
I/O Bus
I/O Inter -face
Device
I/O Inter -face
Device
Accu- mula- tor
General Purpose Registers
DMA Controller

DMA Channels

- The GBA has 4 DMA channels:
  - 0 ← (We don't tend to use this one in 2261)
    - Highest Priority
    - Time Critical Operations
    - Only works with IWRAM (which is a small area)
  - 1 & 2 ← (Stay tuned for latish in the semester)
    - Dedicated to digital sound playback
  - **3 ← (We'll only use this one, for now)**
    - Lowest Priority, highest flexibility
    - General purpose copies, like loading tiles or bitmaps into memory

Note: priority here is about who goes first when multiple DMA attempts occur at the same time -- All DMA channels work at the same speed.

Using DMA
(each channel is configured via 3 registers)

- Source Address Register
  - REG_DMAxSAD (`0x040000B0` for channel 0)
    - (where x is the channel number: 0, 1, 2, 3)
  - The address of the data that will be copied
- Destination Address Register
  - REG_DMAxDAD (`0x040000B4` for channel 0)
  - The address to copy the data to
- Control Register
  - REG_DMAxCNT (DMA control) (`0x040000B8` for channel 0)
  - How much to copy plus some configuration around how to do the copying.

REG_DMAxCNT

- Lower 16 bits specify the amount of memory to transfer
- Upper 16 bits contain other options
  - Turn on a DMA channel
  - When to perform the DMA
  - How the copy source and destination behave
  - How much to copy as each "transfer"/"chunk"
  - Whether or not to throw an interrupt on completion
  - Repeat (or not) when finished
- Can be treated as one 32 bit register, or two 16 bit registers
  - They're all bits, after all!
  - For simplicity we'll treat it as a single 32-bit register.

DMA Control Register Bits

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DMA_ON	IRQ	When			Chunk Size	Repeat	SRC		DST						

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Number of Transfers (max: 65,536)															

This is the number of *chunks* to copy. Not the number of *bytes*. Zero here means do it 2^16 times (since zero made no sense to be an option here)

DMA Control Bits explained

bits	name	description
0-15	N	Number of transfers (where 0 means 2^16).
16-20	NOT USED	AKAIK / according to TONC
21-22	Destination Adjustment bits	
	DMA_DST_INC	00: increment after each transfer (default)
	DMA_DST_DEC	01: decrement after each transfer
	DMA_DST_FIXED	10: none; address is fixed
	DMA_DST_RESET	11: haven't used it yet, but apparently this will increment the destination during the transfer, and reset it to the original value when it's done.
23-24	Source Adjustment bits	
	DMA_SRC_INC	00: increment after each transfer (default)
	DMA_SRC_DEC	01: decrement after each transfer
	DMA_SRC_FIXED	10: none; address is fixed
	DMA_SRC_RESET	11: "forbidden" for source
25	DMA_REPEAT	If set, repeats the copy at each VBlank or HBlank if the DMA timing has been set to those modes.

DMA Control Bits explained

bits	name	bits description
26	CS (Chunk size)	0: Copy by half-word (16 bits) 1: Copy by word (32 bits)
27	NOT USED	
28-29	TM (Timing Mode)	00: Start "immediately" (still a tiny delay before it takes over) 01: Start at VBlank 10: Start at HBlank 11: Start at Refresh - save this one for sound, later
30	I (Interrupt Request)	If set, Raise an interrupt when finished.
31	**En (Enable)**	**Enable the DMA transfer for this channel (turn it on!)**

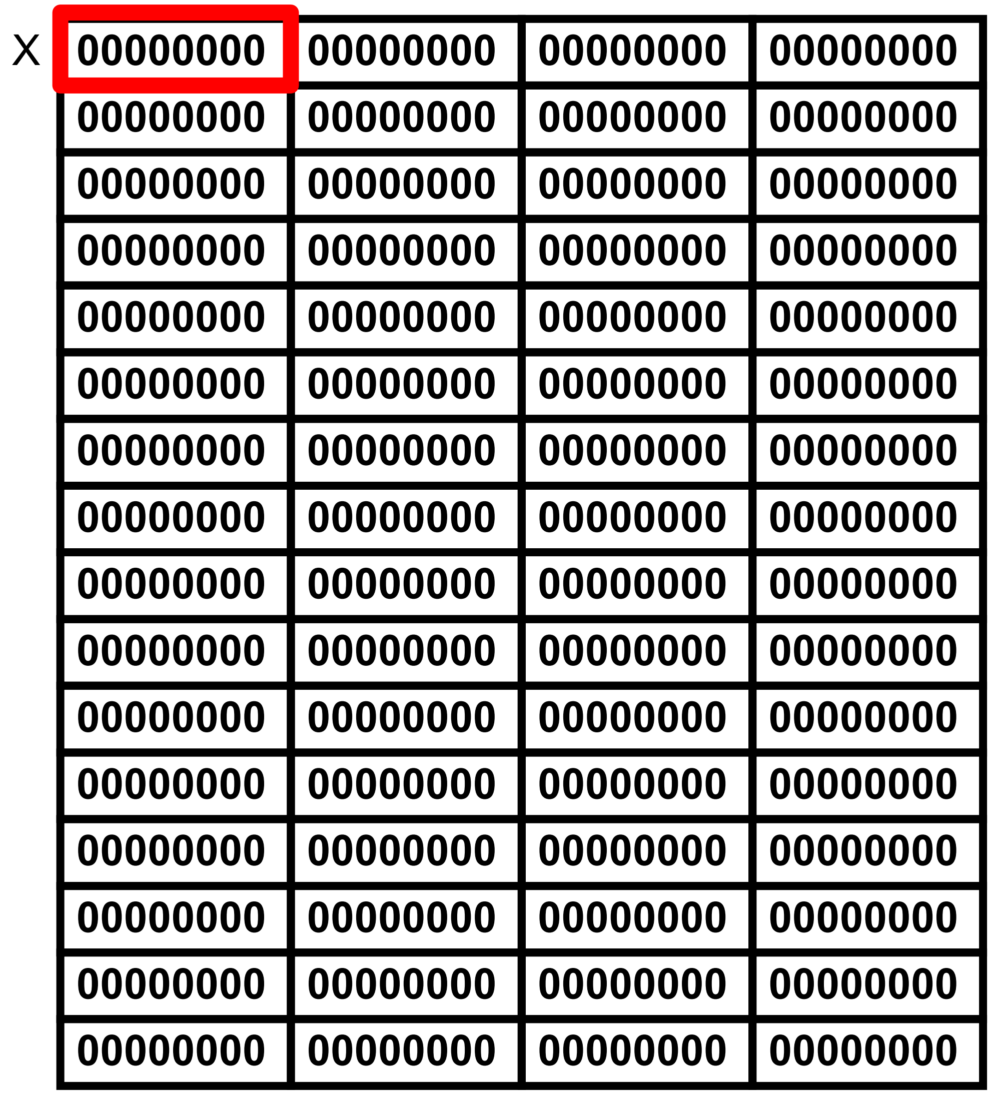
Source/Dest Adjustment

- Increment Source and Dest:
  - Both set to defaults (00)
  - This copies N consecutive chunks from source to dest
  - Ex: drawImage, drawFullScreenImage
- Source Fixed, increment Dest:
  - Source set to (10), Dest set to (00)
  - Fills dest with one value from source
  - Ex: drawRect, fillScreen
- The fixed destination option (10) currently makes no sense at all (lots of overwriting the same spot?!? -- Sound will actually use it later).
- Decrementing just lets you do things backwards (or reverse things, if src. and dest. are going in opposite directions).

Chunk Size and Number

- Ex. Copy an array of 43 shorts
  - chunk size 16 (bits), copy 43 chunks
  - can't (safely) chunk by 32: 21.5 chunks
- Ex. Copy an array of 103 ints
  - chunk size 32, copy 103 chunks
  - chunk size 16, copy 206 chunks (half an int each time)
- Ex. Copy an array of 82 chars
  - chunk size 16, copy 41 chunks (two chars each time)
  - can't chunk by 32, because then you'd want to copy 20.5 chunks

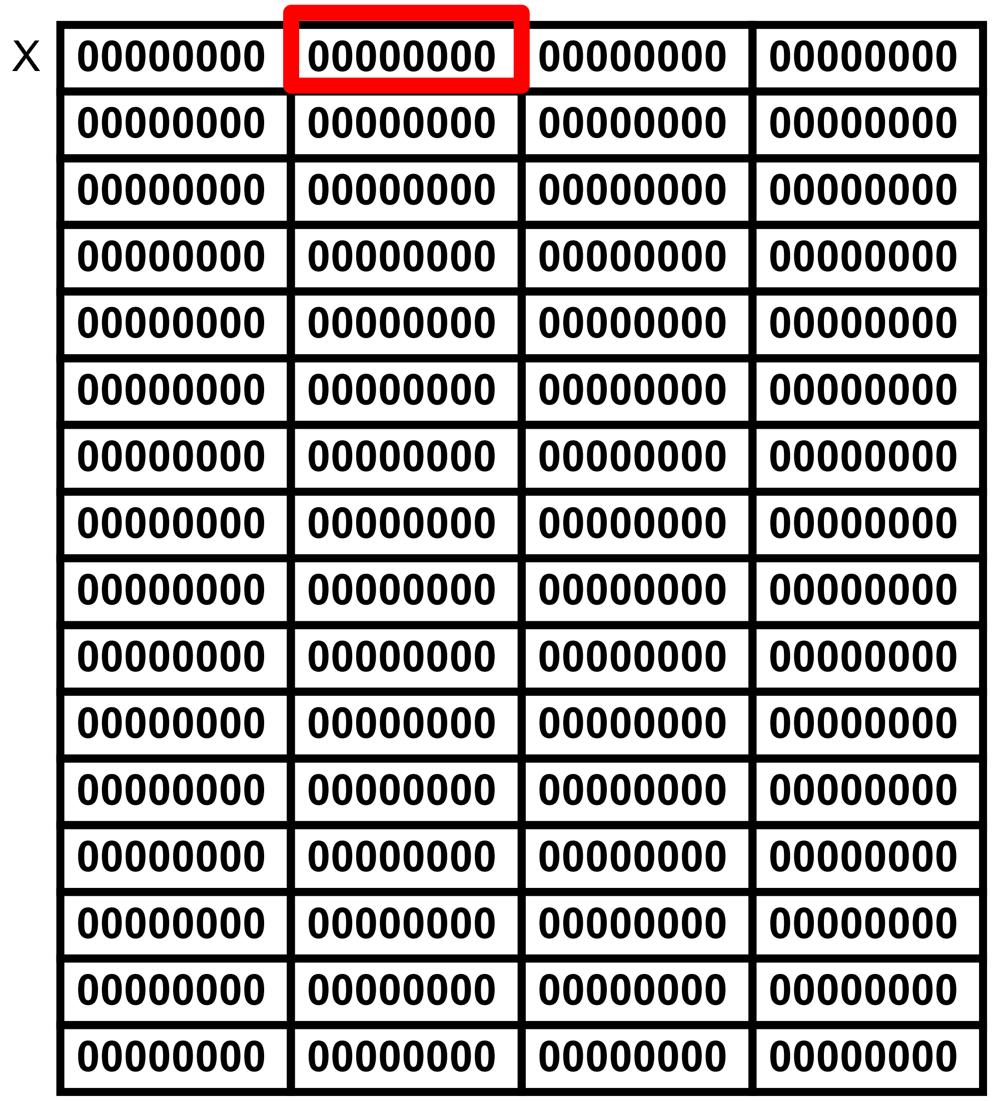
Memory



X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
char *cp = X;
```

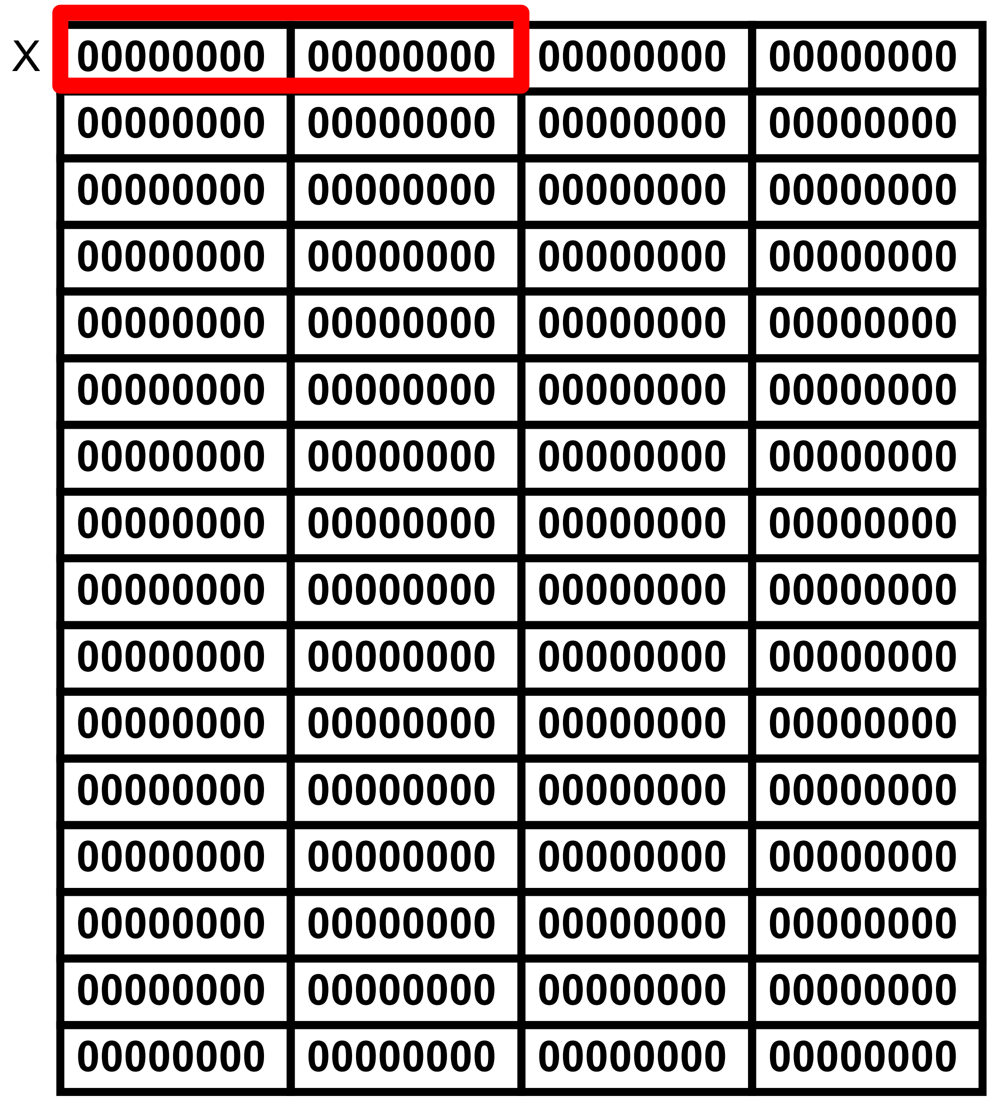
Memory



X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
cp + 1;
```

Memory



X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
short *sp = X;
```

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
sp + 1;
```

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

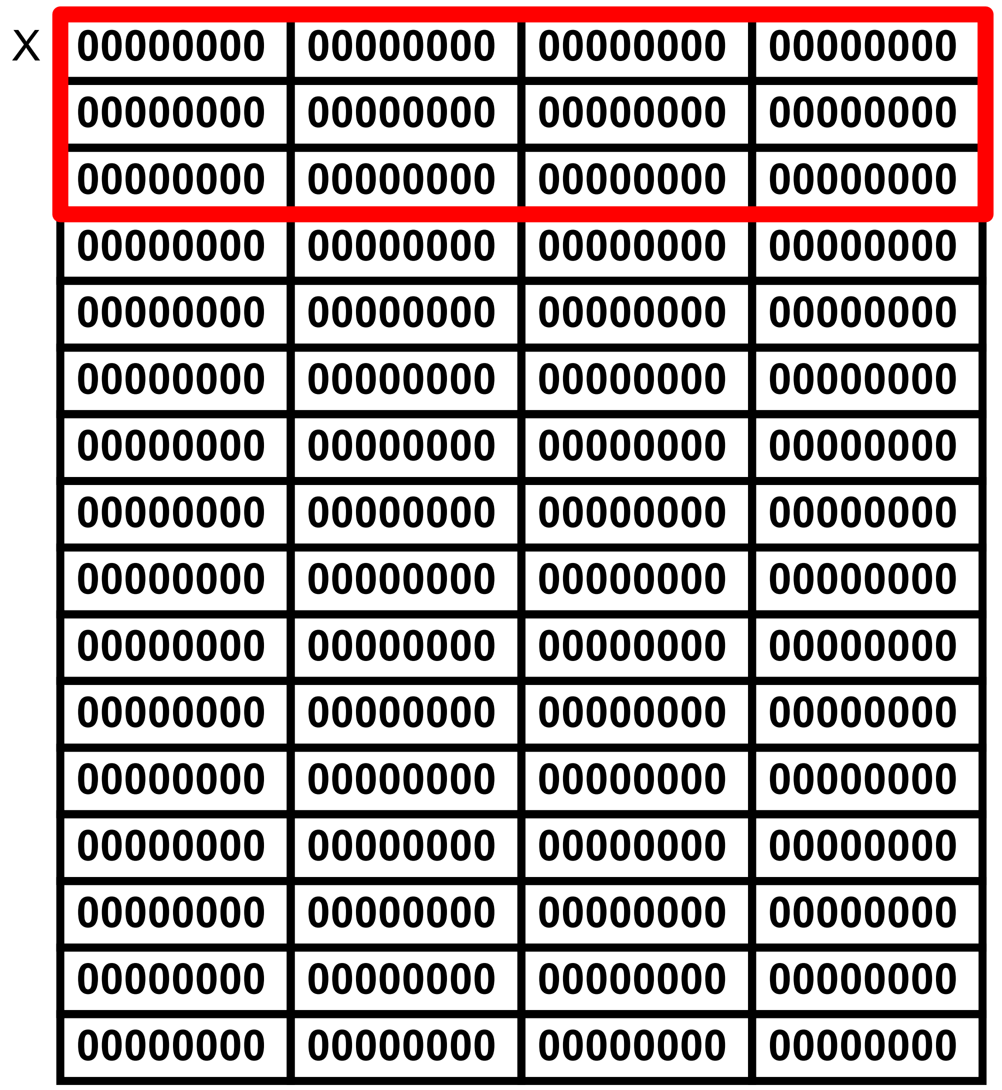
```
int *intptr = X;
```

Memory

X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
intptr + 1;
```

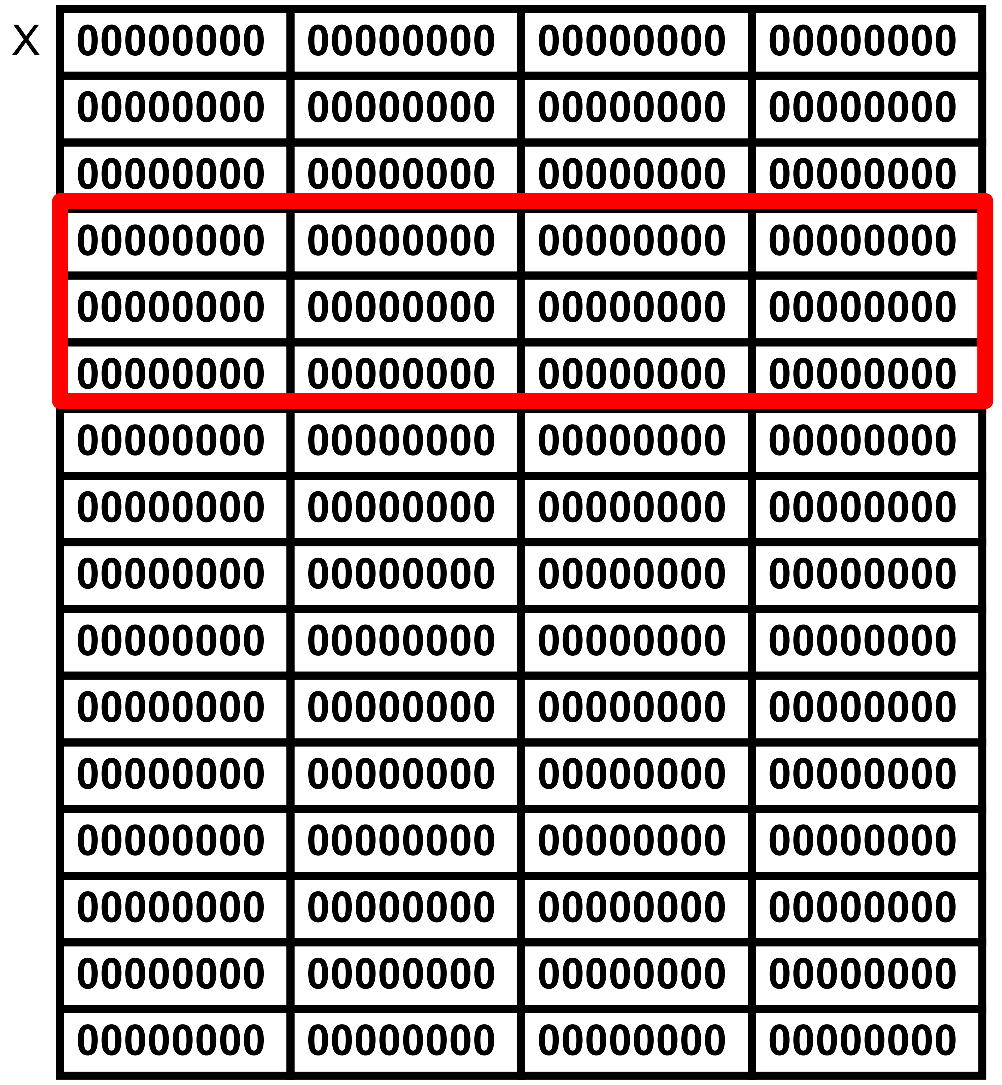
Memory



X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile u32 cnt;
} DMAREC;

DMAREC *dma = X;
```

Memory



X	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000

```
dma + 1;
```

Why are we doing this?

- DMA control registers are all consecutive in memory!
- So we get to leverage pointer arithmetic and structs to let C figure out where everything is for us!

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

DMAREC *dma = (volatile DMAREC *)0x040000B0;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD	dma[0];
	00000000	00000000	00000000	00000000	DMA0DAD	
	00000000	00000000	00000000	00000000	DMA0CNT	
	00000000	00000000	00000000	00000000	DMA1SAD	dma[1];
	00000000	00000000	00000000	00000000	DMA1DAD	
	00000000	00000000	00000000	00000000	DMA1CNT	
	00000000	00000000	00000000	00000000	DMA2SAD	dma[2];
	00000000	00000000	00000000	00000000	DMA2DAD	
	00000000	00000000	00000000	00000000	DMA2CNT	
	00000000	00000000	00000000	00000000	DMA3SAD	dma[3];
	00000000	00000000	00000000	00000000	DMA3DAD	
	00000000	00000000	00000000	00000000	DMA3CNT	

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```c
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

DMAREC *dma = (volatile DMAREC *)0x040000B0;
```

```c
dma[3].src;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

DMAREC *dma = (volatile DMAREC *)0x040000B0;
```

```
dma[3].dst;
```

Memory

0x040000B0	00000000	00000000	00000000	00000000	DMA0SAD
	00000000	00000000	00000000	00000000	DMA0DAD
	00000000	00000000	00000000	00000000	DMA0CNT
	00000000	00000000	00000000	00000000	DMA1SAD
	00000000	00000000	00000000	00000000	DMA1DAD
	00000000	00000000	00000000	00000000	DMA1CNT
	00000000	00000000	00000000	00000000	DMA2SAD
	00000000	00000000	00000000	00000000	DMA2DAD
	00000000	00000000	00000000	00000000	DMA2CNT
	00000000	00000000	00000000	00000000	DMA3SAD
	00000000	00000000	00000000	00000000	DMA3DAD
	00000000	00000000	00000000	00000000	DMA3CNT

```
typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

DMAREC *dma = (volatile DMAREC *)0x040000B0;
```

```
dma[3].cnt;
```

DMA Setup

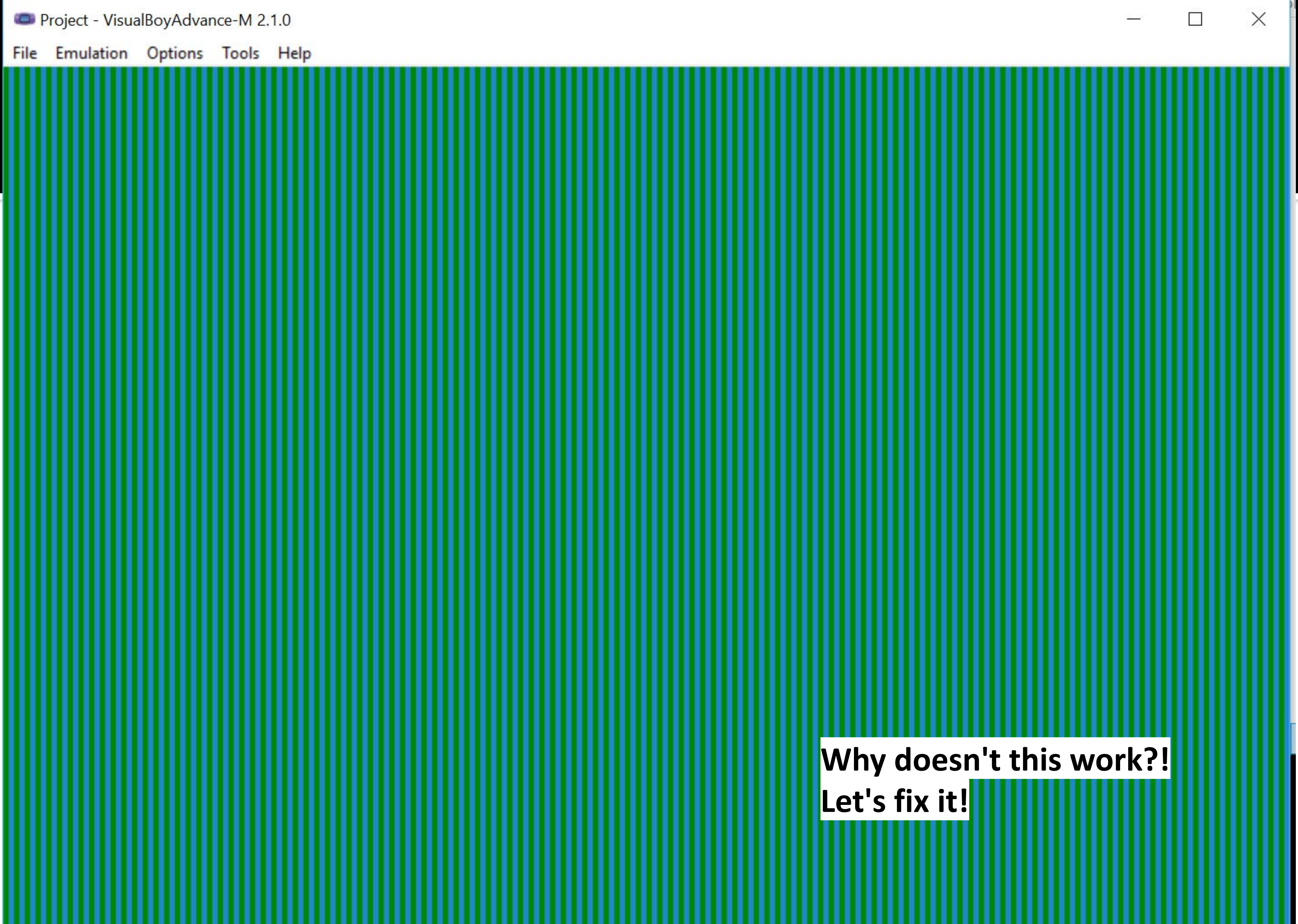
- Map a struct array over the DMA registers

```
typedef struct {
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

#define DMA ((volatile DMAREC*)0x040000b0)
```

Filling the Screen (with demo)

```
#define DMA ((volatile DMAREC*)0x040000b0)

typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

void fillScreen(volatile u16 color){
  DMA[3].cnt = 0;  // No leftover settings breaking things!
  DMA[3].src = &color;
  DMA[3].dst = VIDEO_BUFFER;
  DMA[3].cnt = 1 << 31 | // turn it on!
               1 << 26 | // set chunk size to 32 bits
               1 << 24 | // set src as fixed
               19200;    // 38400 / 2
}
```

```
int main(){
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 i = 0;
  while (1) {
    fillScreen(i);
    i += 127;
  }
}
```


Project - VisualBoyAdvance-M 2.1.0
File Emulation Options Tools Help

Why doesn't this work?!
Let's fix it!

Filling the Screen

```c
#define DMA ((volatile DMAREC*)0x040000b0)

typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

void fillScreen(volatile u16 color){
  DMA[3].cnt = 0;  // No leftover settings breaking things!
  DMA[3].src = &color;
  DMA[3].dst = VIDEO_BUFFER;
  DMA[3].cnt = 1 << 31 | // turn it on!
               // 1 << 26 | // set chunk size to 32 bits
               1 << 24 | // set src as fixed
               38400;
}
```

```c
int main(){
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 i = 0;
  while (1) {
    fillScreen(i);
    i += 127;
  }
}
```

How else could we have fixed it?

Filling the Screen

```
#define DMA ((volatile DMAREC*)0x040000b0)

typedef struct
{
  const volatile void *src;
  volatile void *dst;
  volatile unsigned int cnt;
} DMAREC;

void fillScreen(volatile u16 color){
  // make a copy as the next 2 bytes
  volatile unsigned int temp = color | color << 16;
  DMA[3].cnt = 0;  // No leftover settings breaking things!
  DMA[3].src = &temp;
  DMA[3].dst = VIDEO_BUFFER;
  DMA[3].cnt = 1 << 31 | // turn it on!
               1 << 26 | // set chunk size to 32 bits
               1 << 24 | // set src as fixed
               19200;
}
```

```
int main(){
  REG_DISPCNT = MODE3 | BG2_ENABLE;
  u16 i = 0;
  while (1) {
    fillScreen(i);
    i += 127;
  }
}
```

How else could we have fixed it?

Let's compare to non-DMA speed-wise

(Live Demo DMA vs that for-loop over all 38400 pixels)

(the for-loop is *too slow!*)

Filling a rectangle

Fill row-by-row using DMA for each row.

As long as each row is >10 pixels, it should still be faster than manually drawing every pixel.

When to DMA?

- Copying/filling a lot of data (more than ~10 pixels) with <u>NO LOGIC</u> to the copy.
  - drawRect, fillScreen
  - drawImage
  - arrayCopy
  - arrayReverse

- drawChar with DMA - nope, since you want logic (though you could copyChar with DMA -- if you don't mind getting extra background with it)

Game Boy (DMG) Sound

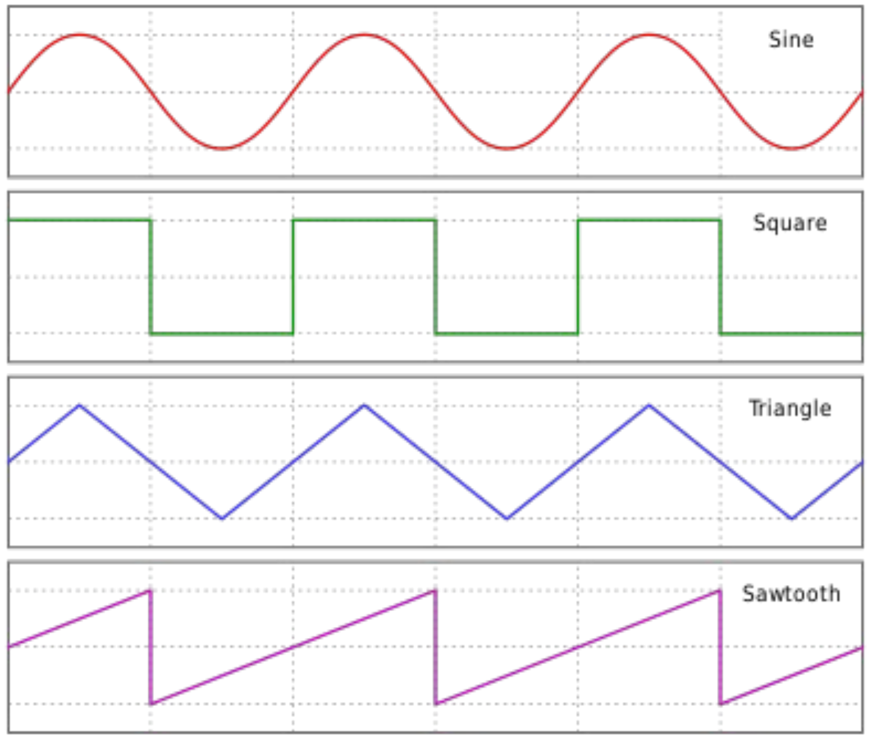
Original Game Boy

- The internal project name for the original Game Boy at Nintendo was *Dot Matrix Game*, and the original hardware serial number was "DMG-01"

- Game Boy resources such as GBATek refer to the original system by this DMG acronym.
- The GBA features a DMG system-on-a-chip, so it doesn't really do any emulation, it just has dedicated hardware.
- The DMG sound hardware is available to the GBA.

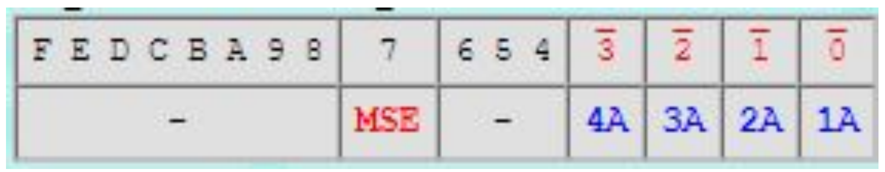
DMG sound has 4 Channels (3 Waveform Generators and a Noise Generator)

- GBA Sound Channel 1 - Tone & Sweep
  - Square wave with optional frequency sweeping
- GBA Sound Channel 2 - Tone
  - Simple Square Waves
- GBA Sound Channel 3 - Wave Output
  - Here you get to define the waveform's shape
- GBA Sound Channel 4 - Noise
  - Used to output noise????
    - This is how digital drumming is done!
  - Can be harsh or soft (think snare vs clap).



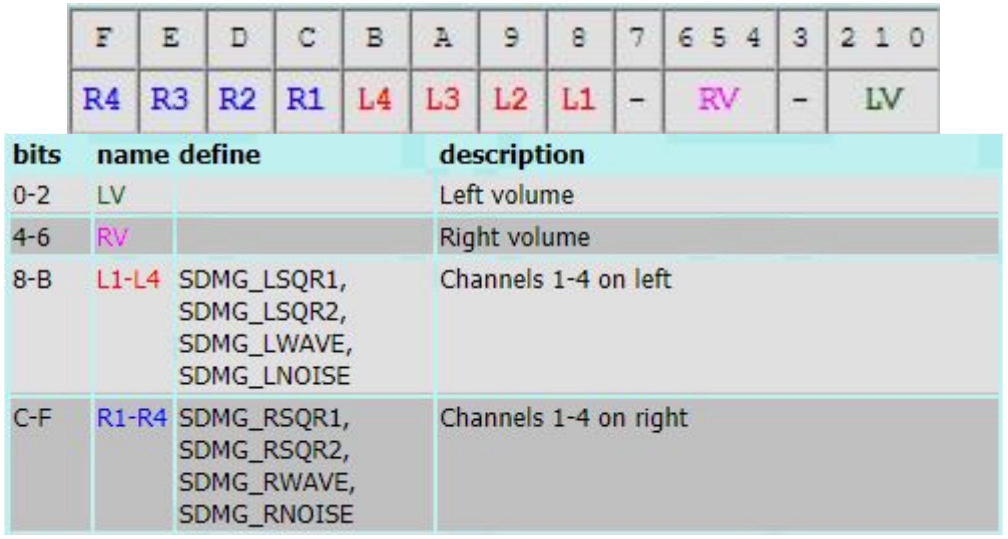
Must-set Registers

```
#define REG_SOUNDCNT_X        *(volatile u16*)0x04000084
#define SND_ENABLED           (1<<7)
// bit 7 is master sound enable for all sounds
// on the GBA.
```



F E D C B A 9 8	7	6 5 4	$\overline{3}$	$\overline{2}$	$\overline{1}$	$\overline{0}$
-	MSE	-	4A	3A	2A	1A

```
#define REG_SOUNDCNT_L        *(volatile u16*)0x04000080


// left/right volume in bits 0-2 and 4-6
// Bits 8-11 enable DMG channels 1-4
// on left speaker, 12-15 on right
```



F	E	D	C	B	A	9	8	7	6 5 4	3	2 1 0
R4	R3	R2	R1	L4	L3	L2	L1	-	RV	-	LV

bits	name	define	description
0-2	LV		Left volume
4-6	RV		Right volume
8-B	L1-L4	SDMG_LSQR1, SDMG_LSQR2, SDMG_LWAVE, SDMG_LNOISE	Channels 1-4 on left
C-F	R1-R4	SDMG_RSQR1, SDMG_RSQR2, SDMG_RWAVE, SDMG_RNOISE	Channels 1-4 on right

```
#define REG_SOUNDCNT_H        *(volatile u16*)0x04000082
// bits 0-1 are DMG sound volume. 00 => 25%, 01 => 50%, 10 => 100%
//                                11 => "forbidden"
```

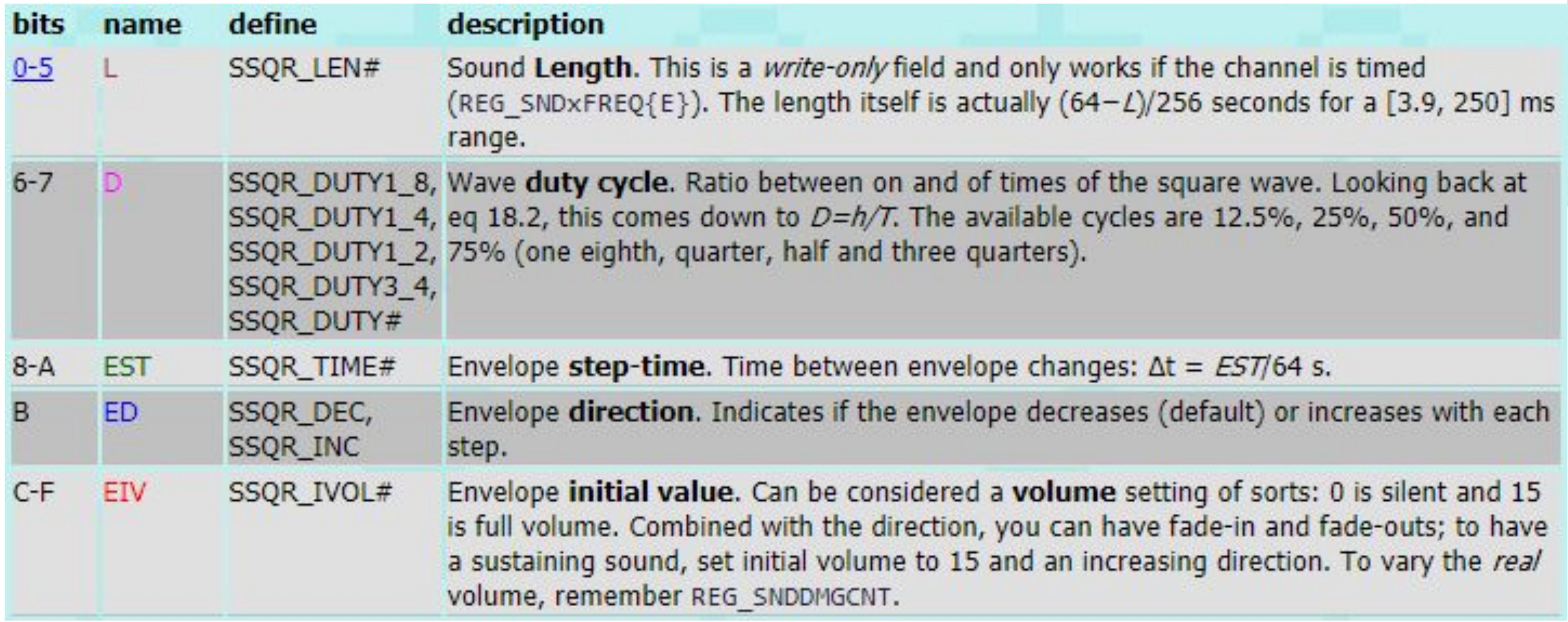
Example setup

```
// Enable sound (Master control)
REG_SOUNDCNT_X = SND_ENABLED;

// Master sound controls for DMG (GameBoy) Sound Generators
REG_SOUNDCNT_L = DMG_VOL_LEFT(5) |
                 DMG_VOL_RIGHT(5) |
                 DMG_SND2_LEFT |
                 DMG_SND2_RIGHT;

REG_SOUNDCNT_H = DMG_MASTER_VOL(2);
```

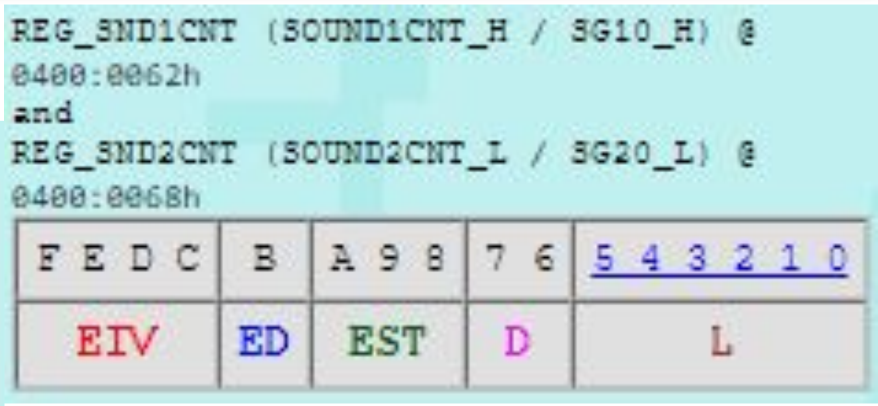
Let's start with Channel 2 (the simplest)

It has two registers for configuring the output:

```
#define REG_SND2CNT   *(volatile u16*)0x04000068
```



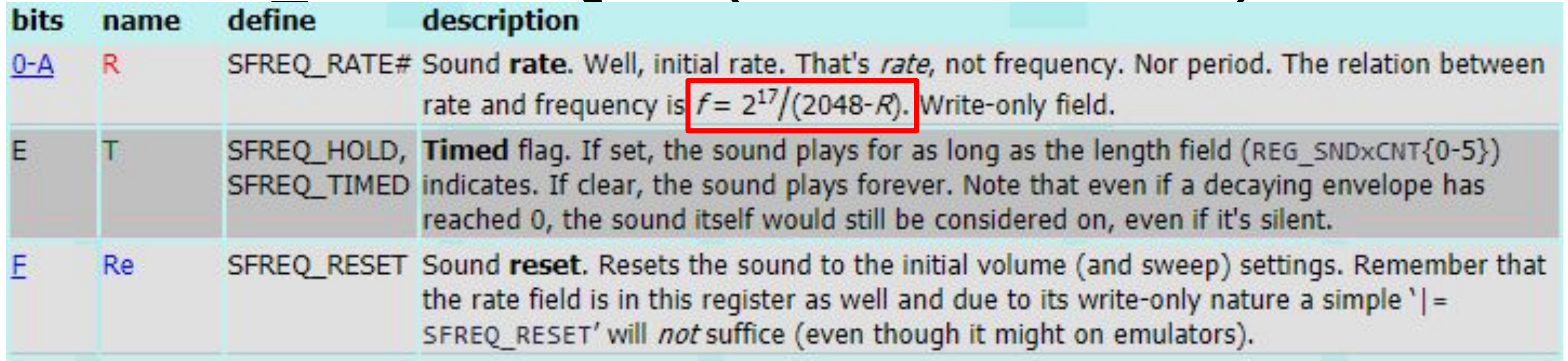
bits	name	define	description
0-5	L	SSQR_LEN#	Sound **Length**. This is a *write-only* field and only works if the channel is timed (REG_SNDxFREQ{E}). The length itself is actually (64−*L*)/256 seconds for a [3.9, 250] ms range.
6-7	D	SSQR_DUTY1_8, SSQR_DUTY1_4, SSQR_DUTY1_2, SSQR_DUTY3_4, SSQR_DUTY#	Wave **duty cycle**. Ratio between on and of times of the square wave. Looking back at eq 18.2, this comes down to *D=h/T*. The available cycles are 12.5%, 25%, 50%, and 75% (one eighth, quarter, half and three quarters).
8-A	EST	SSQR_TIME#	Envelope **step-time**. Time between envelope changes: Δt = *EST*/64 s.
B	ED	SSQR_DEC, SSQR_INC	Envelope **direction**. Indicates if the envelope decreases (default) or increases with each step.
C-F	EIV	SSQR_IVOL#	Envelope **initial value**. Can be considered a **volume** setting of sorts: 0 is silent and 15 is full volume. Combined with the direction, you can have fade-in and fade-outs; to have a sustaining sound, set initial volume to 15 and an increasing direction. To vary the *real* volume, remember REG_SNDDMGCNT.



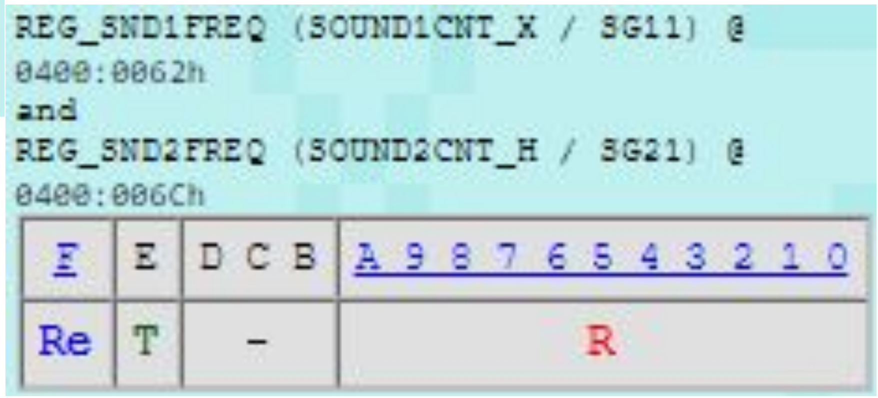
REG_SND1CNT (SOUND1CNT_H / SG10_H) @ 0400:0062h
and
REG_SND2CNT (SOUND2CNT_L / SG20_L) @ 0400:0068h

F E D C	B	A 9 8	7 6	5 4 3 2 1 0
EIV	ED	EST	D	L

```
#define REG_SND2FREQ  *(volatile u16*)0x0400006C
```



bits	name	define	description
0-A	R	SFREQ_RATE#	Sound **rate**. Well, initial rate. That's *rate*, not frequency. Nor period. The relation between rate and frequency is $\boxed{f = 2^{17}/(2048\text{-}R)}$. Write-only field.
E	T	SFREQ_HOLD, SFREQ_TIMED	**Timed** flag. If set, the sound plays for as long as the length field (REG_SNDxCNT{0-5}) indicates. If clear, the sound plays forever. Note that even if a decaying envelope has reached 0, the sound itself would still be considered on, even if it's silent.
F	Re	SFREQ_RESET	Sound **reset**. Resets the sound to the initial volume (and sweep) settings. Remember that the rate field is in this register as well and due to its write-only nature a simple '|= SFREQ_RESET' will *not* suffice (even though it might on emulators).

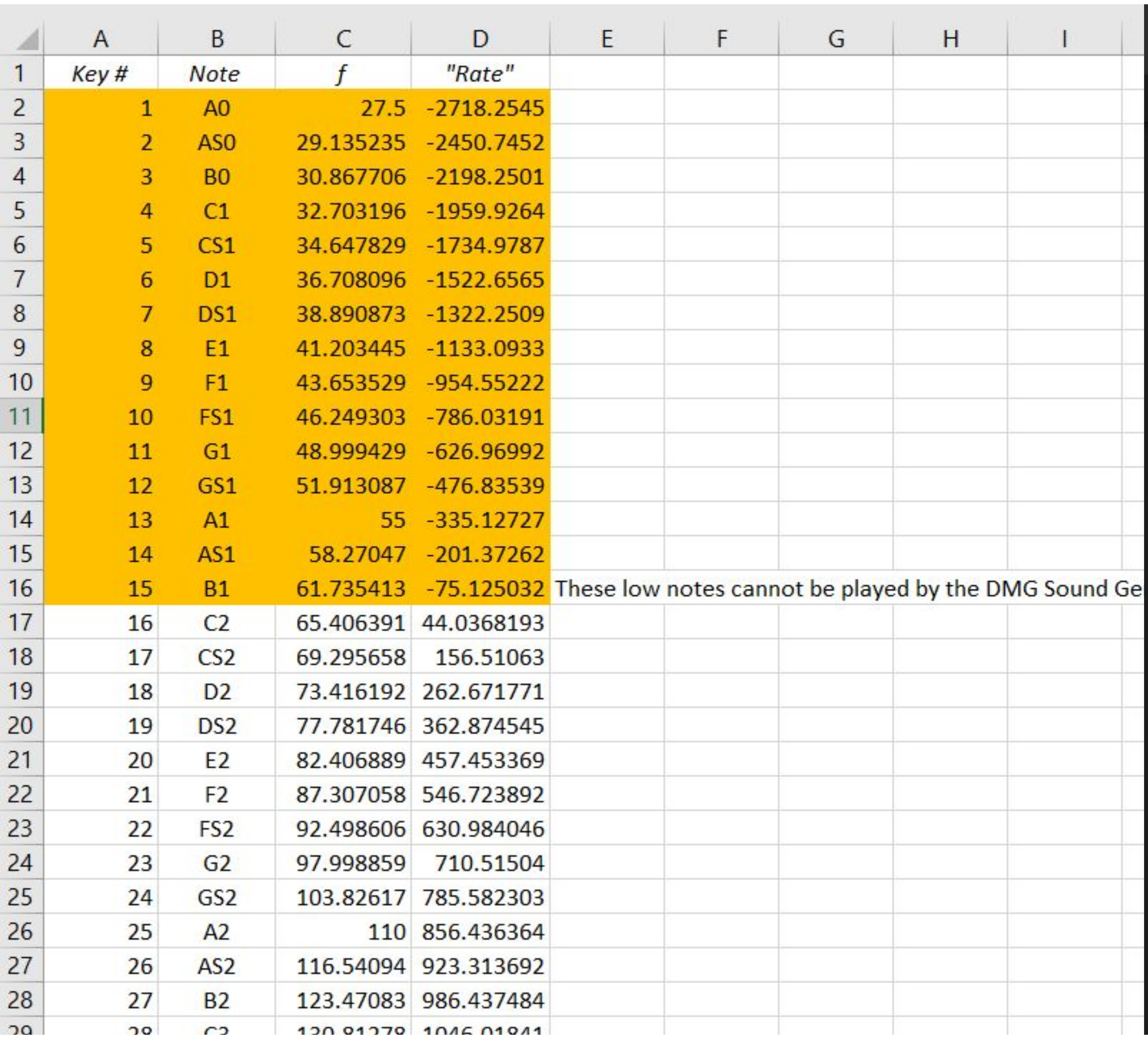


REG_SND1FREQ (SOUND1CNT_X / SG11) @ 0400:0062h
and
REG_SND2FREQ (SOUND2CNT_H / SG21) @ 0400:006Ch

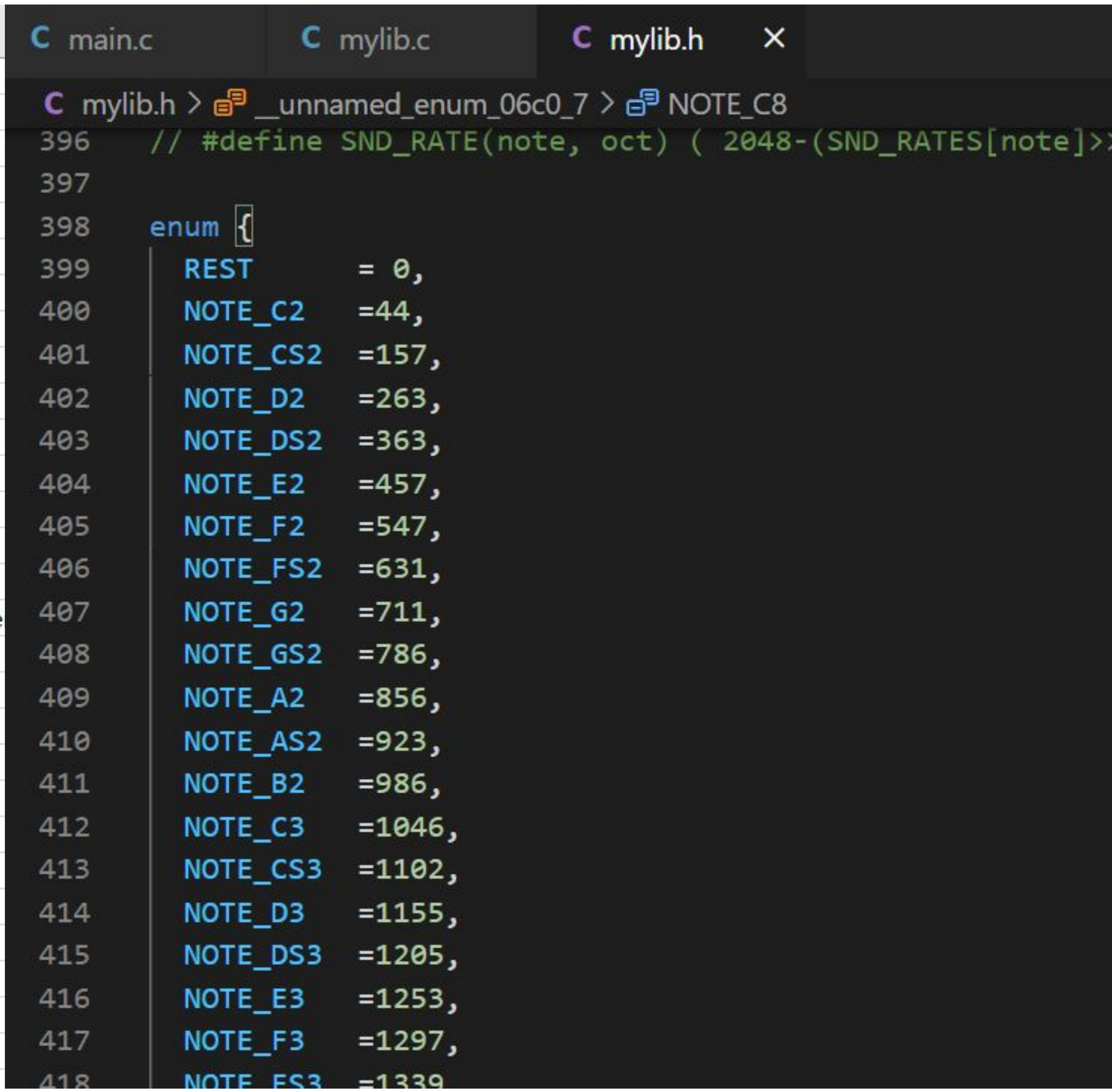
F	E	D C B	A 9 8 7 6 5 4 3 2 1 0
Re	T	-	R

Frequency vs "Rate"

I calculated them in Excel and then just made another look-up table:



	A	B	C	D	E
1	*Key #*	*Note*	*f*	*"Rate"*	
2	1	A0	27.5	-2718.2545	
3	2	AS0	29.135235	-2450.7452	
4	3	B0	30.867706	-2198.2501	
5	4	C1	32.703196	-1959.9264	
6	5	CS1	34.647829	-1734.9787	
7	6	D1	36.708096	-1522.6565	
8	7	DS1	38.890873	-1322.2509	
9	8	E1	41.203445	-1133.0933	
10	9	F1	43.653529	-954.55222	
11	10	FS1	46.249303	-786.03191	
12	11	G1	48.999429	-626.96992	
13	12	GS1	51.913087	-476.83539	
14	13	A1	55	-335.12727	
15	14	AS1	58.27047	-201.37262	
16	15	B1	61.735413	-75.125032	These low notes cannot be played by the DMG Sound Ge
17	16	C2	65.406391	44.0368193	
18	17	CS2	69.295658	156.51063	
19	18	D2	73.416192	262.671771	
20	19	DS2	77.781746	362.874545	
21	20	E2	82.406889	457.453369	
22	21	F2	87.307058	546.723892	
23	22	FS2	92.498606	630.984046	
24	23	G2	97.998859	710.51504	
25	24	GS2	103.82617	785.582303	
26	25	A2	110	856.436364	
27	26	AS2	116.54094	923.313692	
28	27	B2	123.47083	986.437484	
29	28	C3	130.81278	1046.01841	



```c
// #define SND_RATE(note, oct) ( 2048-(SND_RATES[note]>

enum {
    REST        = 0,
    NOTE_C2    =44,
    NOTE_CS2   =157,
    NOTE_D2    =263,
    NOTE_DS2   =363,
    NOTE_E2    =457,
    NOTE_F2    =547,
    NOTE_FS2   =631,
    NOTE_G2    =711,
    NOTE_GS2   =786,
    NOTE_A2    =856,
    NOTE_AS2   =923,
    NOTE_B2    =986,
    NOTE_C3    =1046,
    NOTE_CS3   =1102,
    NOTE_D3    =1155,
    NOTE_DS3   =1205,
    NOTE_E3    =1253,
    NOTE_F3    =1297,
    NOTE_FS3   =1339
```

Helpful macros

```
#define REG_SND1SWEEP      *(volatile u16*)0x04000060
#define REG_SND1CNT        *(volatile u16*)0x04000062
#define REG_SND1FREQ       *(volatile u16*)0x04000064

#define REG_SND2CNT        *(volatile u16*)0x04000068
#define REG_SND2FREQ       *(volatile u16*)0x0400006C

#define REG_SND3SEL        *(volatile u16*)0x04000070
#define REG_SND3CNT        *(volatile u16*)0x04000072
#define REG_SND3FREQ       *(volatile u16*)0x04000074

#define REG_SND4CNT        *(volatile u16*)0x04000078
#define REG_SND4FREQ       *(volatile u16*)0x0400007C

// Channel 3 Wave Pattern RAM (2 banks!!)
#define REG_SND3_WAV       *(volatile u16*)0x04000090

#define DMG_SND1_LEFT          (1 << 8)
#define DMG_SND2_LEFT          (1 << 9)
#define DMG_SND3_LEFT          (1 << 10)
#define DMG_SND4_LEFT          (1 << 11)

#define DMG_SND1_RIGHT         (1 << 12)
#define DMG_SND2_RIGHT         (1 << 13)
```

```
#define DMG_SND3_RIGHT         (1 << 14)
#define DMG_SND4_RIGHT         (1 << 15)

// n: [0-7]
#define DMG_VOL_LEFT(n)        (((n) & 7) << 0)
#define DMG_VOL_RIGHT(n)       (((n) & 7) << 4)

// n: [0-15]
#define DMG_ENV_VOL(n)         (((n) & 15) << 12)

// n: [0-7]
#define DMG_STEP_TIME(n)       (((n) & 7) << 8)

#define DMG_DIRECTION_DECR     (0 << 11)
#define DMG_DIRECTION_INCR     (1 << 11)

#define DMG_DUTY_12            (0 << 6)
#define DMG_DUTY_25            (1 << 6)
#define DMG_DUTY_50            (2 << 6)
#define DMG_DUTY_75            (3 << 6)

// n: [0-2]
#define DMG_MASTER_VOL(n)      ((n) % 3)
#define SND_RESET              (1<<15)
```

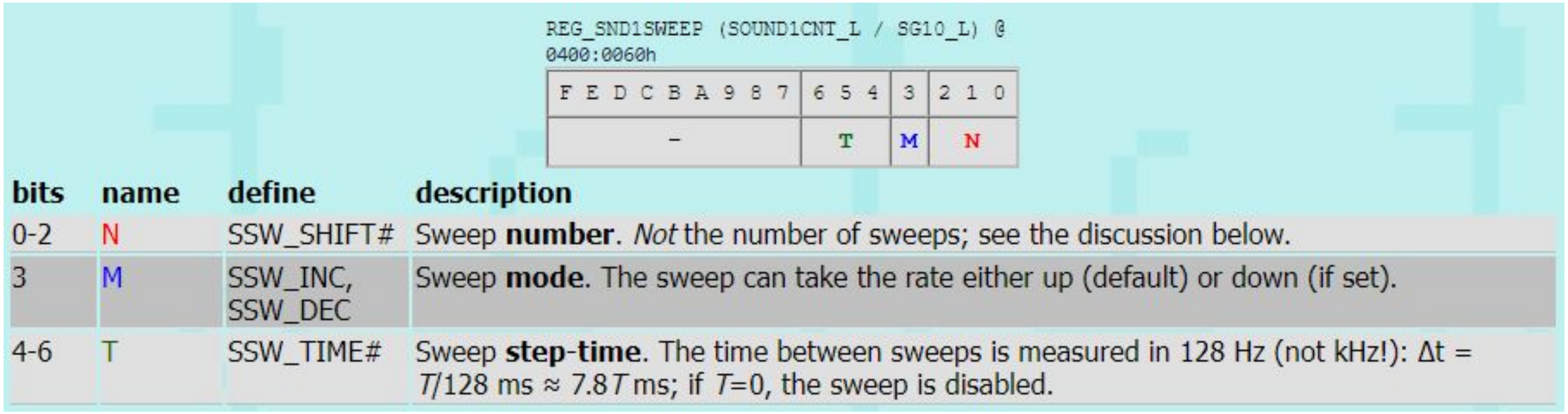
Simple struct + array + loop in interruptHandler to get something playing

```
typedef struct noteWithDuration {
  u16 note;      // From NOTES enum, by name
  u16 duration;  // in vBlanks
} NoteWithDuration;

NoteWithDuration song[] = {
  {REST, 60},
  {NOTE_C4, 30},
  {NOTE_C4, 30},
  {NOTE_G4, 30},
  {NOTE_G4, 30},
  {NOTE_A4, 15},
  {NOTE_A4, 15},
  {NOTE_A4, 15}
}
```

```
int main()
  while(1) {
    if (BUTTON_PRESSED(BUTTON_A)){
      if (song[note].note != REST) {
        REG_SND2CNT = DMG_ENV_VOL(1) | DMG_DIRECTION_DECR |
DMG_STEP_TIME(7) | DMG_DUTY_50;
        REG_SND2FREQ = song[note].note | SND_RESET | DMG_FREQ_TIMED;
      }
      note++;
      if (note >= songLength) {
        note = 0;
      }
    }
    oldbuttons = BUTTONS;
    waitForVBlank();
  }
}
```

Channel 1 = Channel2 + Sweep

From Tonc:



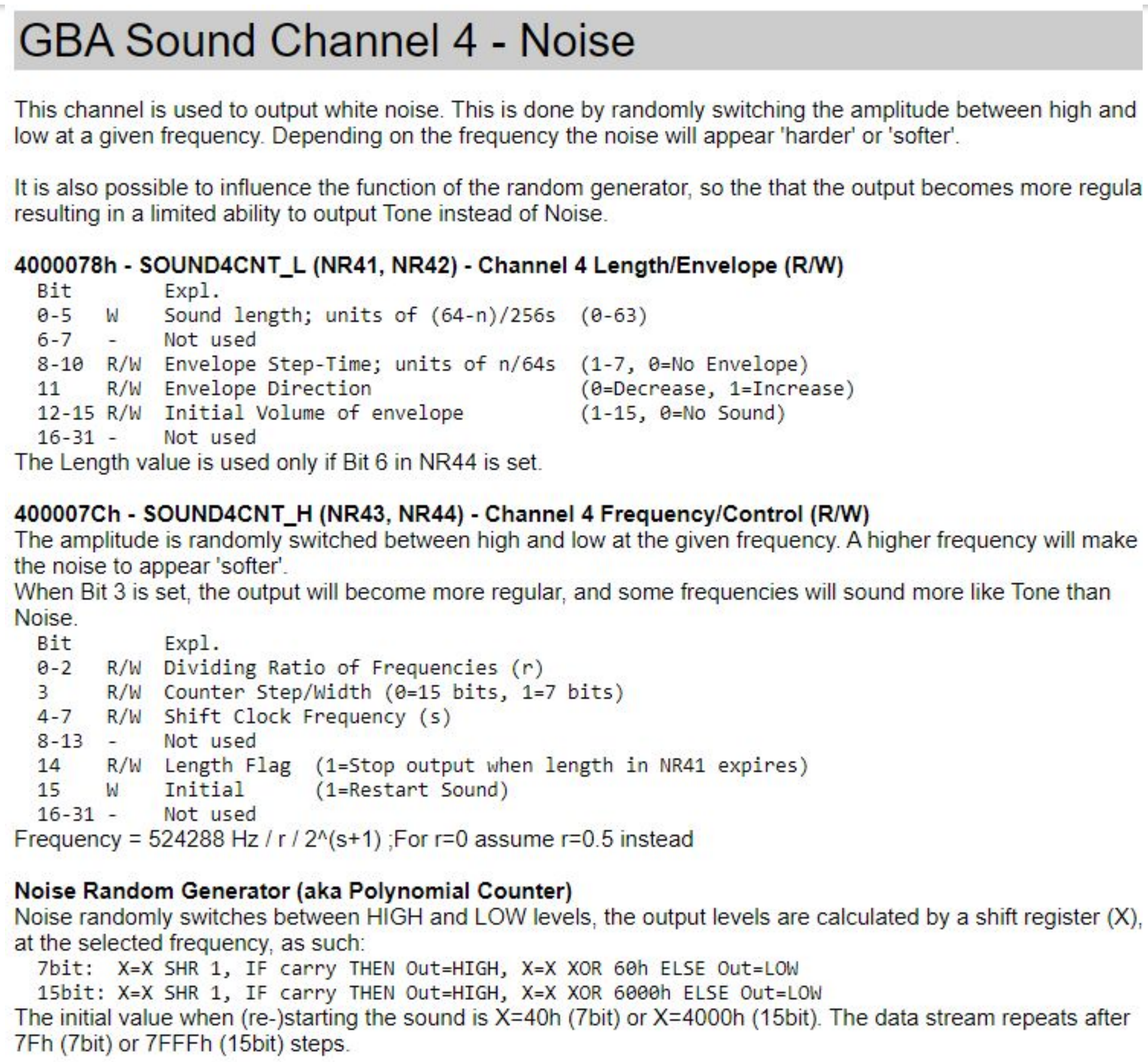
REG_SND1SWEEP (SOUND1CNT_L / SG10_L) @ 0400:0060h

F E D C B A 9 8 7	6 5 4	3	2 1 0
-	T	M	N

bits	name	define	description
0-2	N	SSW_SHIFT#	Sweep **number**. *Not* the number of sweeps; see the discussion below.
3	M	SSW_INC, SSW_DEC	Sweep **mode**. The sweep can take the rate either up (default) or down (if set).
4-6	T	SSW_TIME#	Sweep **step-time**. The time between sweeps is measured in 128 Hz (not kHz!): Δt = *T*/128 ms ≈ 7.8*T* ms; if *T*=0, the sweep is disabled.

Channel 4: Noise

From GBATek:



GBA Sound Channel 4 - Noise

This channel is used to output white noise. This is done by randomly switching the amplitude between high and low at a given frequency. Depending on the frequency the noise will appear 'harder' or 'softer'.

It is also possible to influence the function of the random generator, so the that the output becomes more regula resulting in a limited ability to output Tone instead of Noise.

4000078h - SOUND4CNT_L (NR41, NR42) - Channel 4 Length/Envelope (R/W)

Bit		Expl.	
0-5	W	Sound length; units of (64-n)/256s	(0-63)
6-7	-	Not used	
8-10	R/W	Envelope Step-Time; units of n/64s	(1-7, 0=No Envelope)
11	R/W	Envelope Direction	(0=Decrease, 1=Increase)
12-15	R/W	Initial Volume of envelope	(1-15, 0=No Sound)
16-31	-	Not used	

The Length value is used only if Bit 6 in NR44 is set.

400007Ch - SOUND4CNT_H (NR43, NR44) - Channel 4 Frequency/Control (R/W)
The amplitude is randomly switched between high and low at the given frequency. A higher frequency will make the noise to appear 'softer'.
When Bit 3 is set, the output will become more regular, and some frequencies will sound more like Tone than Noise.

Bit		Expl.
0-2	R/W	Dividing Ratio of Frequencies (r)
3	R/W	Counter Step/Width (0=15 bits, 1=7 bits)
4-7	R/W	Shift Clock Frequency (s)
8-13	-	Not used
14	R/W	Length Flag (1=Stop output when length in NR41 expires)
15	W	Initial (1=Restart Sound)
16-31	-	Not used

Frequency = 524288 Hz / r / 2^(s+1) ;For r=0 assume r=0.5 instead

Noise Random Generator (aka Polynomial Counter)
Noise randomly switches between HIGH and LOW levels, the output levels are calculated by a shift register (X), at the selected frequency, as such:

```
  7bit:  X=X SHR 1, IF carry THEN Out=HIGH, X=X XOR 60h ELSE Out=LOW
  15bit: X=X SHR 1, IF carry THEN Out=HIGH, X=X XOR 6000h ELSE Out=LOW
```

The initial value when (re-)starting the sound is X=40h (7bit) or X=4000h (15bit). The data stream repeats after 7Fh (7bit) or 7FFFh (15bit) steps.