First Steps to C Mastery

Today's Lecture

First steps towards C mastery

- Arithmetic: adders, wraparound
- Software: hallmarks of good software
- Pointers, structs, and memory
- More on Heap Allocator

Some fun: MIDI

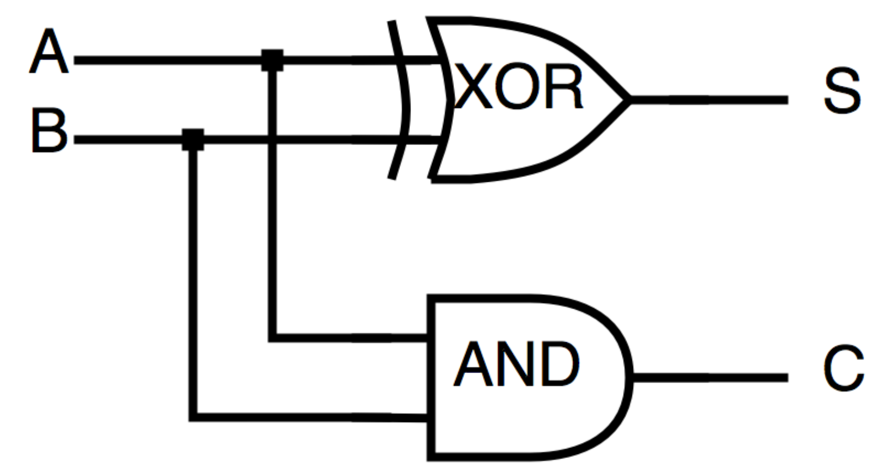
1-Bit Adder

Add 2 1-bit numbers

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10

Add 2 1-bit numbers (Half Adder)

a	b	sum
0	0	00
0	1	01
1	0	01
1	1	10



bit 0 of sum: S = a^b

bit 1 of sum: C = a&b

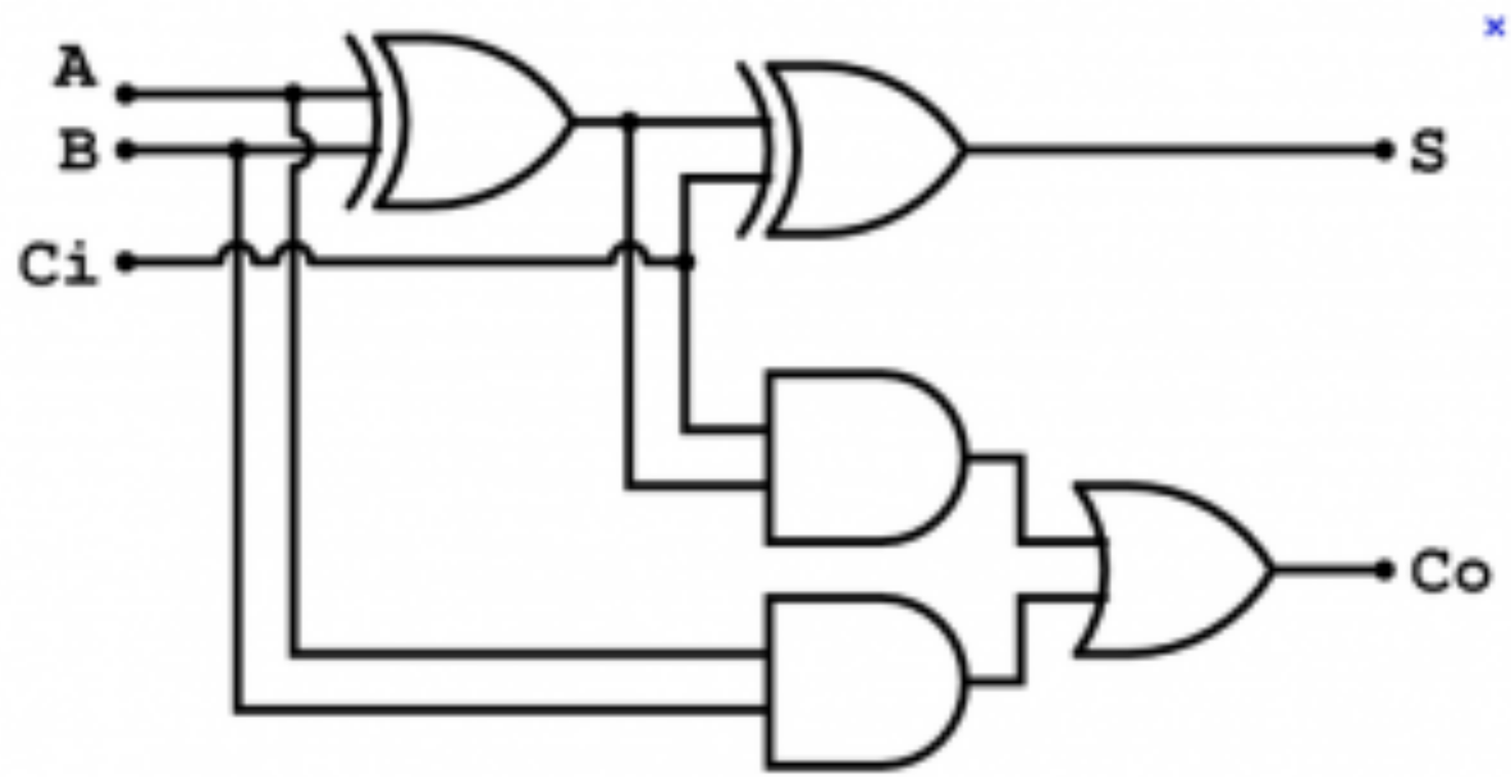
Have reduced addition to bitwise, logical operations!

Add 3 1-bit numbers

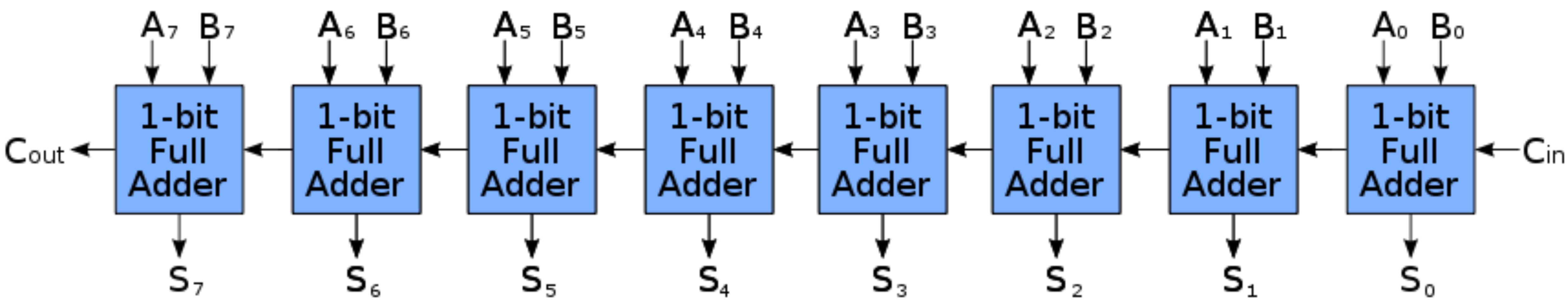
a	b	c	=	c	s
0	0	0		0	0
0	1	0		0	1
1	0	0		0	1
1	1	0		1	0
0	0	1		0	1
0	1	1		1	0
1	0	1		1	0
1	1	1		1	1

Add 3 1-bit numbers (Full Adder)

a	b	ci	=	co	s
0	0	0		0	0
0	1	0		0	1
1	0	0		0	1
1	1	0		1	0
0	0	1		0	1
0	1	1		1	0
1	0	1		1	0
1	1	1		1	1



8-bit adder


A7 B7
A6 B6
A5 B5
A4 B4
A3 B3
A2 B2
A1 B1
A0 B0
Cout
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
1-bit Full Adder
Cin
S7
S6
S5
S4
S3
S2
S1
S0

But What Type Is It?

```
unsigned int b = 2;
int a = -2;

if(a > b)
    printf("a > b\n");
else
    printf("b > a\n");
```

```
int b = 2;
int a = -2;

if(a > b)
    printf("a > b\n");
else
    printf("b > a\n");
```

But What Type Is It?

```
unsigned int b = 2;
int a = -2;

if(a > b)
    printf("a > b\n");
else
    printf("b > a\n");
```

```
int b = 2;
int a = -2;

if(a > b)
    printf("a > b\n");
else
    printf("b > a\n");
```

C converts a into `unsigned`

Conversions Everywhere

```c
#include <stdio.h>
#include <limits.h>
#include <assert.h>

int main(void) {
  assert(sizeof(unsigned char)==1);

  unsigned char uc1 = 0xff;
  unsigned char uc2 = 0;

  if(~uc1 == uc2) {
    printf("%hhx == %hhx\n", ~uc1, uc2);
  } else {
    printf("%hhx != %hhx\n", ~uc1, uc2);
  }
  return 0;
}
```

Operand Conversion

	u8	u16	u32	u64	i8	i16	i32	i64
u8	i32	i32	u32	u64	i32	i32	i32	i64
u16	i32	i32	u32	u64	i32	i32	i32	i64
u32	u32	u32	u32	u64	u32	u32	u32	i64
u64	u64	u64	u64	u64	u64	u64	u64	u64
i8	i32	i32	u32	u64	i32	i32	i32	i64
i16	i32	i32	u32	u64	i32	i32	i32	i64
i32	i32	i32	u32	u64	i32	i32	i32	i64
i64	i64	i64	i64	u64	i64	i64	i64	i64

code/wraparound

- In two’s complement, when you exceed the maximum value of int (2,147,483,647), you “wrap around” to negative numbers:

- Here is the link after Google upgraded to 64-bit integers:

Writing Good Systems Software

```
void serial_init() {
    unsigned int ra;

    // Configure the UART
    PUT32(AUX_ENABLES, 1);
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_CNTL_REG, 0); // Default RTS/CTS
    PUT32(AUX_MU_LCR_REG, 3); // Put in 8 bit mode
    PUT32(AUX_MU_MCR_REG, 0); // Default RTS/CTS auto flow control
    PUT32(AUX_MU_IER_REG, 0); // Clear FIFO
    PUT32(AUX_MU_IIR_REG, 0xC6); // Baudrate
    PUT32(AUX_MU_BAUD_REG, 270); // Baudrate

    // Configure the GPIO lines
    ra = GET32(GPFSEL1);
    ra &= ~(7 << 12); //gpio14
    ra |= 2 << 12;    //alt5
    ra &= ~(7 << 15); //gpio15
    ra |= 2 << 15;    //alt5
    PUT32(GPFSEL1,ra);
    PUT32(GPPUD,0);
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, (1 << 14) | (1 << 15));
    for (ra = 0; ra < 150; ra++) dummy(ra);
    PUT32(GPPUDCLK0, 0);

    // Enable the serial port (both RX and TX)
    PUT32(AUX_MU_CNTL_REG, 3);
}
```

```
void uart_init(void) {
    int *aux = (int*)AUX_ENABLES;;

    *aux = AUX_ENABLE; // turn on mini-uart

    uart->ier  = 0;
    uart->cntl = 0;
    uart->lcr  = MINI_UART_LCR_8BIT;
    uart->mcr  = 0;
    uart->ier  = 0;
    uart->iir  = MINI_UART_IIR_RX_FIFO_CLEAR |
                 MINI_UART_IIR_RX_FIFO_ENABLE |
                 MINI_UART_IIR_TX_FIFO_CLEAR |
                 MINI_UART_IIR_TX_FIFO_ENABLE;

    uart->baud =  270; // baud rate ((250,000,000/115200)/8)-1 = 270

    gpio_set_function(GPIO_TX, GPIO_FUNC_ALT5);
    gpio_set_function(GPIO_RX, GPIO_FUNC_ALT5);

    uart->cntl = MINI_UART_CNTL_TX_ENABLE |
                 MINI_UART_CNTL_RX_ENABLE;
}
```

Well-written software is easy for someone to read and understand.

Well-written software is easy for someone to read and understand.

Code that is easier to understand has fewer bugs.

Long comments != easy to read and understand.

Understand at the line, function, file, and structural levels.

Lesson: Imagine someone else has to fix a bug in your code: what should it look like make this easier? *Hint: be a section leader, you'll have to read student code and you'll learn a lot.*

Systems Code Is Terse But Unforgiving

Systems Code Is Terse But Unforgiving

Next week in lab you will write a "PS/2 scan code reader" to read keyboard input: if any part of it is wrong, you won't read scan codes. It's only 20-30 lines of code!

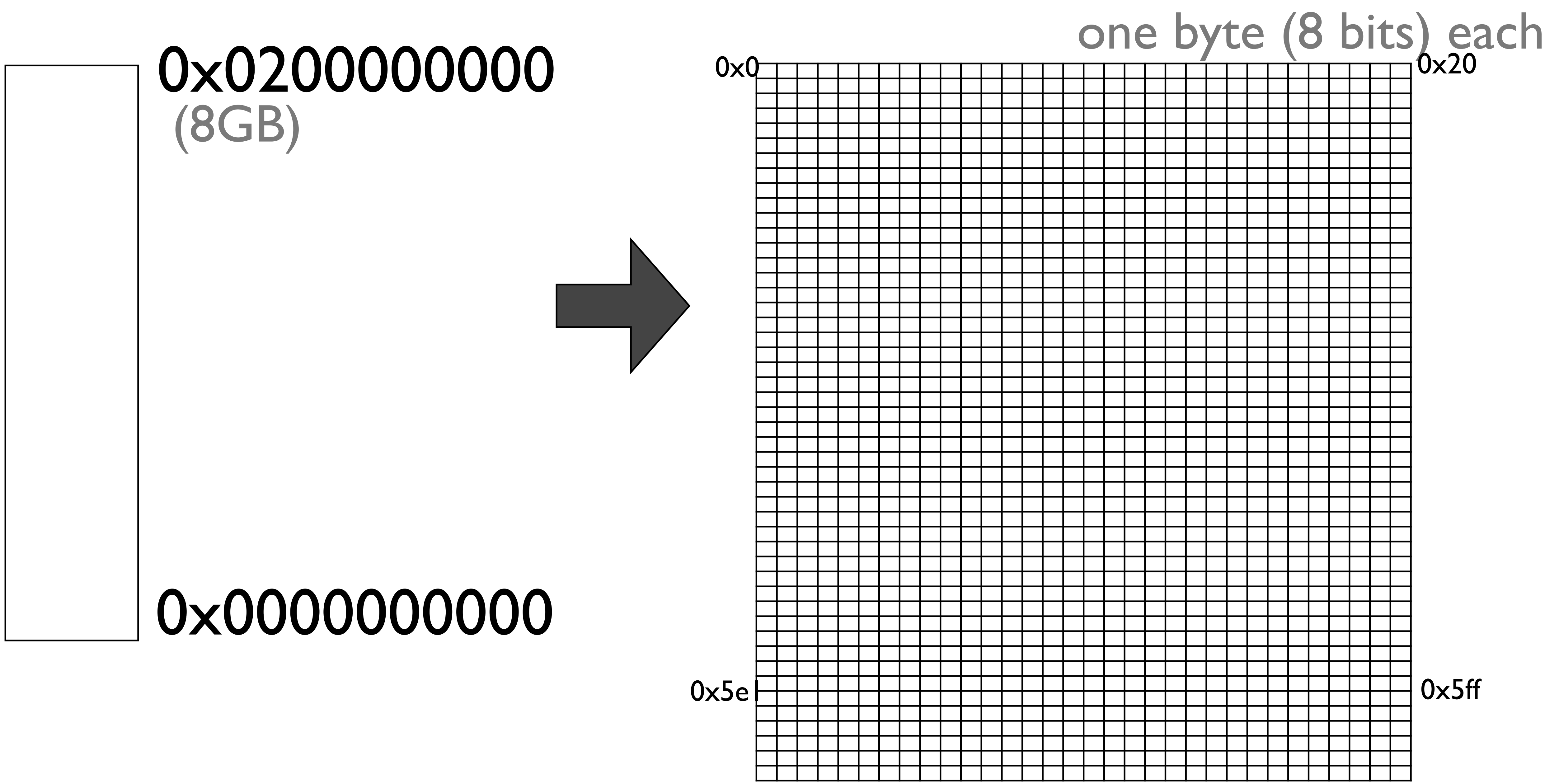
The mailbox code you'll use for the frame buffer is ~10 lines of code: we once debugged it for 9 hours.

Lesson: if you know exactly what you have to do it can take only minutes; throwing away and rewriting can often be *faster*. Sunk cost fallacy!

```
void uart_init(void) {
    int *aux = (int*)AUX_ENABLES;;

    *aux = AUX_ENABLE; // turn on mini-uart

    uart->ier  = 0;
    uart->cntl = 0;
    uart->lcr  = MINI_UART_LCR_8BIT;
    uart->mcr  = 0;
    uart->ier  = 0;
    uart->iir  = MINI_UART_IIR_RX_FIFO_CLEAR |
                 MINI_UART_IIR_RX_FIFO_ENABLE |
                 MINI_UART_IIR_TX_FIFO_CLEAR |
                 MINI_UART_IIR_TX_FIFO_ENABLE;

    uart->baud =  270; // baud rate ((250,000,000/115200)/8)-1 = 270

    gpio_set_function(GPIO_TX, GPIO_FUNC_ALT5);
    gpio_set_function(GPIO_RX, GPIO_FUNC_ALT5);

    uart->cntl = MINI_UART_CNTL_TX_ENABLE | MINI_UART_CNTL_RX_ENABLE;;
}
```

Pointers, Structures, Etc.

Computer Memory



Endianness

Multibyte words: how do you arrange the bytes?

1,024 = 0x0400 =

?	?

Little endian: least significant byte is at lowest address

- Makes most sense from an addressing/computational standpoint

0x00	0x04

Big endian: most significant byte is at lowest address

- Makes most sense to a human reader

0x04	0x00

“Arrays are Pointers”

Sort of true. Sometimes.

When are arrays and pointers different?

- A pointer is a location in memory storing an address (you can change the pointer)
- An array is a location in memory storing data (you can change the *elements* of an array, but not the array itself)

code/pointers

C structs

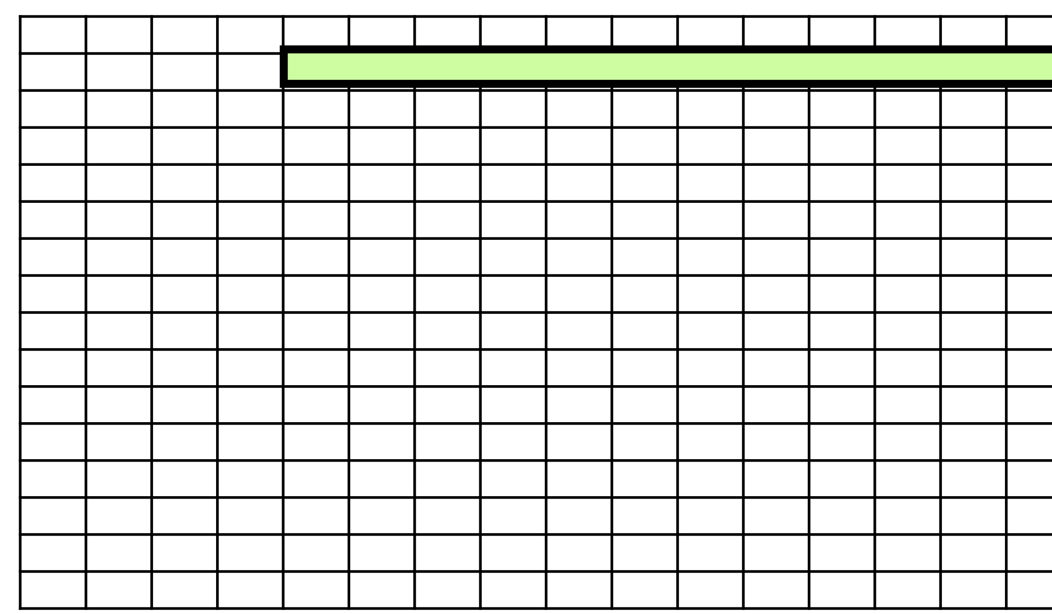
```
struct data {
  unsigned char fields;
  unsigned int num_changes;
  char name[MAX_NAME + 1];
  unsigned short references;
  unsigned short links;
}
```

How big is this structure?
How is it laid out in memory (on an ARM)?

C structs

A C struct is just a shorthand and convenient way to allocate memory and describe offsets from a pointer.

```
struct data {
  unsigned char fields;
  unsigned int num_changes;
  char name[4];
  unsigned short references;
  unsigned short links;
};

struct data* d = …;
d->fields = 0;      // d + 0
d->num_changes = 0; // d + 4
d->references = 0;  // d + 12
d->links = 0;       // d + 14
```

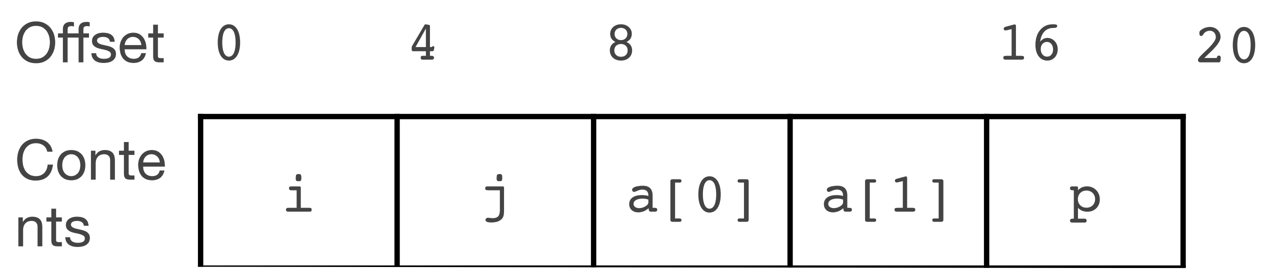


Structures

Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 4-byte `int` pointer, for a total of 20 bytes:



- The numbers along the top of the diagram are the byte offsets of the fields from the beginning of the structure.
- Struct layout is guaranteed to be in the order you defined, however the exact location is dependent on alignment in the memory system (ARM requires 4-byte alignment).
- Note that the array is embedded in the structure.
- To access the fields, the compiler generates code that adds the field offset to the address of the structure.

Structures

Example:

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};
```

This structure has four fields: two 4-byte values of type `int`, a two-element array of type `int`, and an 4-byte `int` pointer, for a total of 20 bytes:

Offset	0	4	8		16	20
Contents	i	j	a[0]	a[1]	p	

- Let's look at the following code in gdb:

```
struct rec r;
r.i = 1;
r.j = 2;
r.a[0] = 3;
r.a[1] = 4;
r.p = (int *)0x8000;
```

What's this about alignment?

Example:

Let's look at a different struct.

```
struct withchar {
    int i;
    int j;
    char a;
    char b;
    int k;
};
```

Offset	0	4	8	12	16
Contents	i	j	ab	p	

- It looks like there is some waste! There are two unused bytes at locations 10 and 11 -- this is because in order to meet alignment requirements, the compiler had to put k at location 12, and it couldn't put it at location 10 (not divisible by 4).

- Let's look at the following code in gdb:

```
    struct withchar wc;
    wc.i = 1;
    wc.j = 2;
    wc.a = 'a';
    wc.b = 'b';
    wc.k = 3;
```

- Remember: the compiler *must* put your struct in order, but it also must meet alignment requirements.

Laying a struct onto memory

```
struct rec {
    int i;
    int j;
    int a[2];
    int *p;
};

void main(void)
{
    char arr[1024];
    for (int i=0; i < 1024 / 4; i++) {
        *((int *)arr + i) = 0xdeadbeef; // fill with deadbeef
    }
    char *randomLocation = &arr[64];
    ((struct rec *)randomLocation)->i = 5;
    ((struct rec *)randomLocation)->j = 6;
    ((struct rec *)randomLocation)->a[0] = 7;
    ((struct rec *)randomLocation)->a[1] = 8;
    ((struct rec *)randomLocation)->p = (int *)0x8000;
```

- We can put a struct anywhere, as long as we align it to a 4-byte boundary

Laying a struct onto memory

```
(gdb) x/40x arr
0x7fffbb4:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffbc4:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffbd4:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffbe4:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffbf4:     0x00000005     0x00000006     0x00000007     0x00000008
0x7fffc04:     0x00008000     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffc14:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffc24:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffc34:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
0x7fffc44:     0xdeadbeef     0xdeadbeef     0xdeadbeef     0xdeadbeef
(gdb)
```

- On Monday, we discussed the *implicit free list*, which is one way to create a heap allocator. It uses what is called a **block header** to hold the information. In the assignment diagrams page (http://cs107e.github.io/assignments/assign4/images/block_hdr/) there is a potential header defined as follows*:

```
struct header {
    size_t payload_size;
    int status;       // 0 if free, 1 if in use
};
```

- The block header is actually *stored in the same memory area as the payload, and it generally precedes the payload.*
- Let's take a look at a series of mallocs, frees, and reallocs and see what happens in this heap.
- We must align each pointer returned to the malloc caller on a 4-byte boundary (even if we will waste some bytes).

*feel free to define a header more efficiently!

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- When the heap is initialized, assume there are 96 bytes starting at location 0x9000. Note in the diagram, 00 = 0x9000, 04 = 0x9004, 08 = 0x9008, etc.
- None of the data in the heap is initialized (represented by "?")
- We need to set up the heap so we can use it, so we first put in a header that tells us that the whole heap is free.

- We need to set up the heap so we can use it, so we first put in a header that tells us that the whole heap is free.

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
88	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

The shaded area is our header, and it says that there are 88 bytes free in the heap? Why 88?

How can we use our struct to set this value? Assume that `heap_start` is a `void *` pointer to the start of our heap:

```
(struct header *)heap_start->payload_size = 88;
(struct header *)heap_start->status = 0;
```

In your malloc.c code, you will want to do this without the 88 constant. If you want to know the size of a header, you can simply write:

```
sizeof(struct block)
```

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	
b	
c	
d	
e	

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
88	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	
b	
c	
d	
e	

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
88	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9008
b	
c	
d	
e	

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	64	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- We change the header at 0x9000 to 16 for the allocation, and set the status to 1 ("used").
- We then add another header for the next block, and set its value to 64 and the status to 0 ("free")
- We then pass back 0x9008 to the calling function.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9008
b	
c	
d	
e	

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	64	0	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- We first check the initial header at location 0x9008 and find that we cannot allocate space there (it is used), so we jump ahead by 16 bytes from the end of our header to find the next header. We find that there are 64 bytes free, enough for our allocation of 8, so we place the block there...

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	
d	
e	

96 bytes

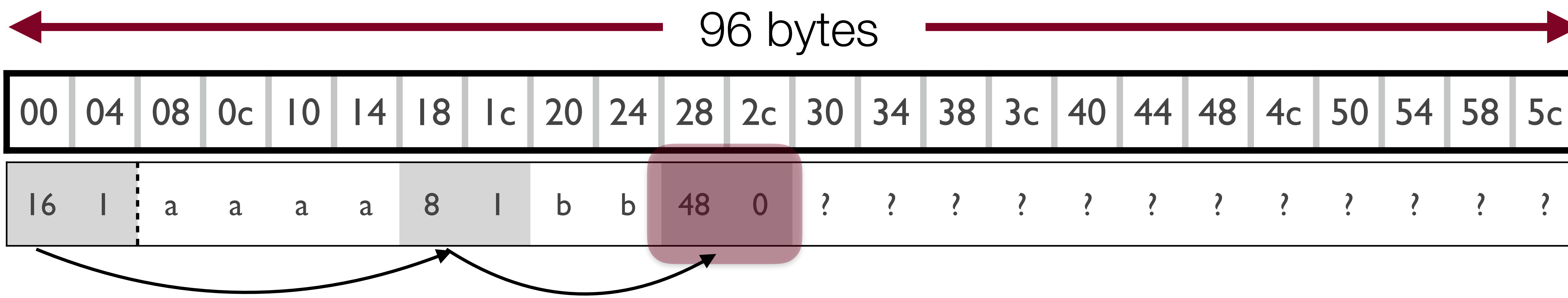
00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	48	0	?	?	?	?	?	?	?	?	?	?	?	?

- We have now updated b's header, and put a header after it to designate the rest of the free block.
- We return 0x9020 to the calling function.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9008
b	0x9020
c	
d	
e	



- We walk the heap to find a free block that fits for c, as well.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	**0x9030**
d	
e	

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	1	c	c	c	c	24	0	?	?	?	?	?	?

- We return 0x9030 to the calling function.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9008
b	0x9020
c	0x9030
d	0x9048
e	0x9058

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	1	c	c	c	c	8	1	d	d	8	1	e	e

- We repeat the process for d and e.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	**0x9030**
d	**0x9048**
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	1	c	c	c	c	8	1	d	d	8	1	e	e

- Now we need to free d.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	**0x9030**
d	
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	1	c	c	c	c	8	0	?	?	8	1	e	e

- Now we need to free d.
- We mark d as free, and then see if the follow-on block is free (it isn't).

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	**0x9030**
d	
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	1	c	c	c	c	8	0	?	?	8	1	e	e

- Now we need to free c.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	
d	
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	16	0	?	?	?	?	8	0	?	?	8	1	e	e

- We mark c as free, and then check to see if there is a free block to the right -- there is!

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9008
b	0x9020
c	
d	
e	0x9058

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	32	0	?	?	?	?	?	?	?	?	8	1	e	e

- We mark c as free, and then check to see if there is a free block to the right -- there is!
- We "coalesce" the blocks because they are next to each other.
- We gain back a bit of space, and we also get a bigger block to allocate later.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	
d	
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	32	0	?	?	?	?	?	?	?	?	8	1	e	e

- We now try to realloc a, and we want 20 bytes. Do we have room in the location? No (if we did, we would simply update the size of a's block and return the same pointer)

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	**0x9008**
b	**0x9020**
c	
d	
e	**0x9058**

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	1	a	a	a	a	8	1	b	b	32	0	?	?	?	?	?	?	?	?	8	1	e	e

- Instead, we need to walk the list, and we find the 32 free block.

- Let's look at the following sets of mallocs, frees, and reallocs:

```
a = malloc(16);
b = malloc(8);
c = malloc(16);
d = malloc(8);
e = malloc(8);
free(d);
free(c);
realloc(a,20);
```

	Value
a	0x9030
b	0x9020
c	
d	
e	0x9058

96 bytes

00	04	08	0c	10	14	18	1c	20	24	28	2c	30	34	38	3c	40	44	48	4c	50	54	58	5c
16	0	?	?	?	?	8	1	b	b	32	1	a	a	a	a	a	a	a	a	8	1	e	e

- Instead, we need to walk the list, and we find the 32 free block.
- We *copy the contents of a* to the free block, free the original, and then return 0x9030.
- By the way: we couldn't just allocate 20 in this case, because we wouldn't have enough room for a header after a!

MIDI

Digital != Analog

Our CPU is generating a square pulse wave

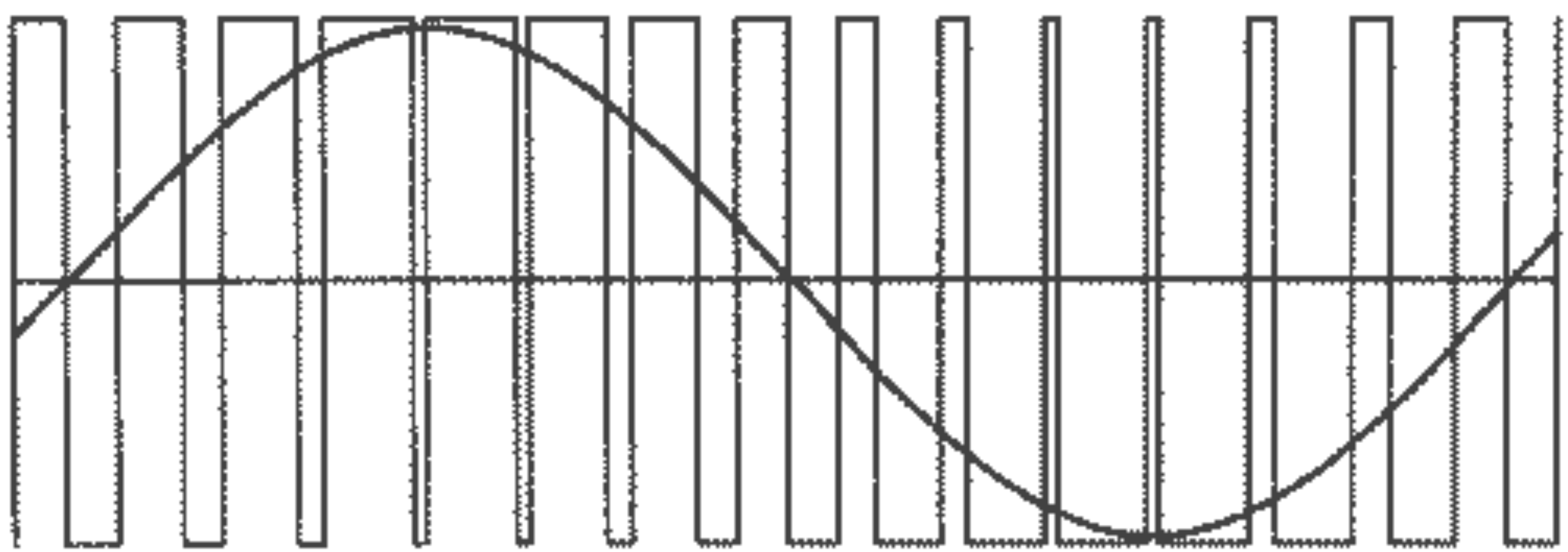
Interacts with electrical components: changing electric field, impedance, capacitance, etc.

- Note: cannot actually send pulse wave

These details are why building high-frequency circuits (e.g., radio, HDMI) requires very careful engineering

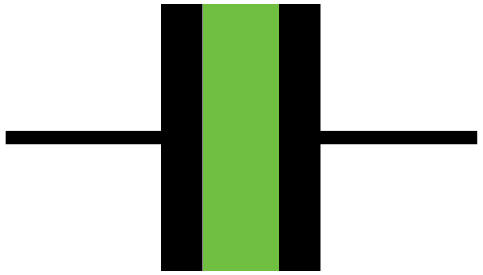
PWM to the Rescue!

Can simulate continuous values with fast enough PWM clocking: need hardware help

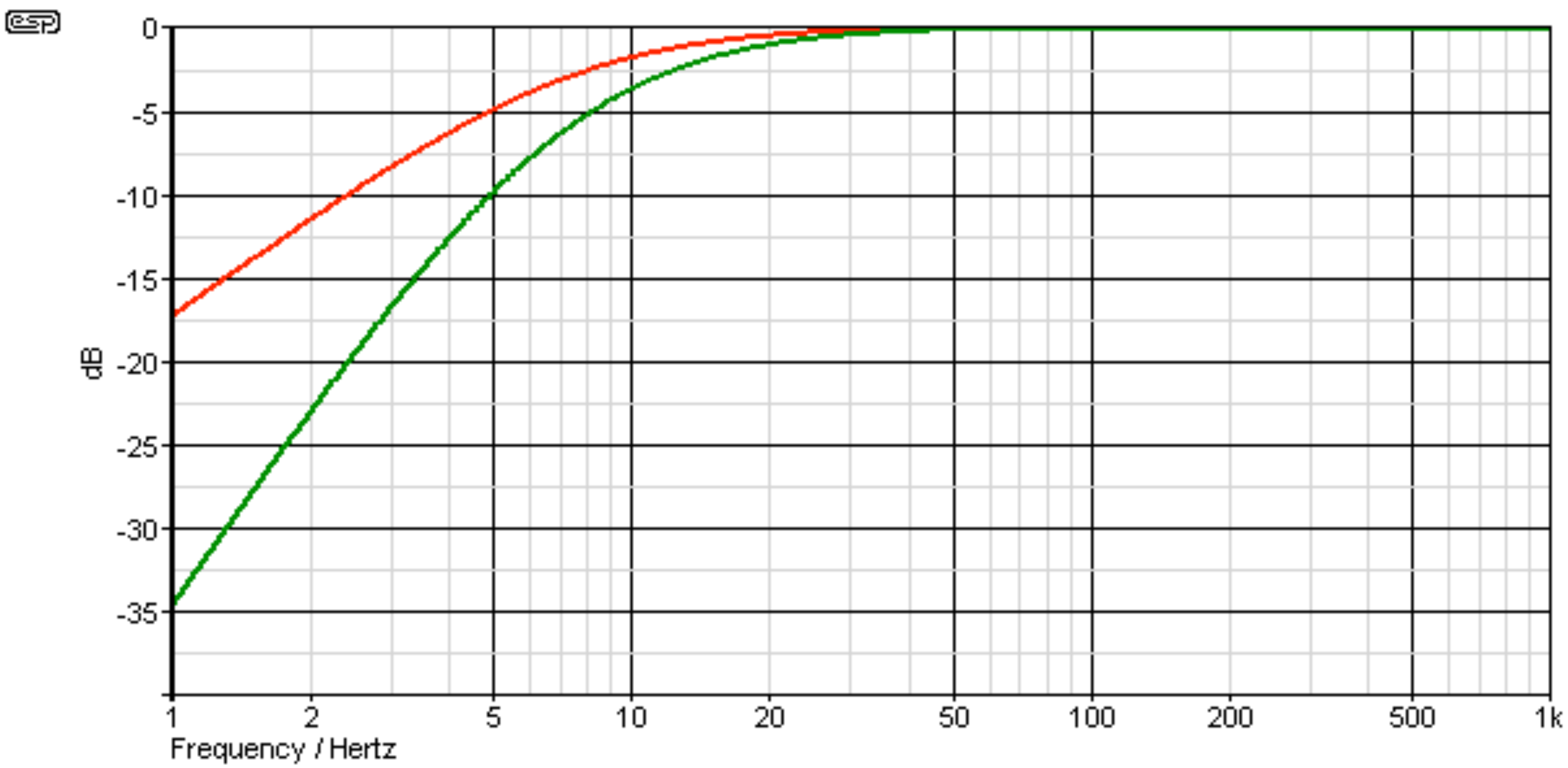


Capacitors

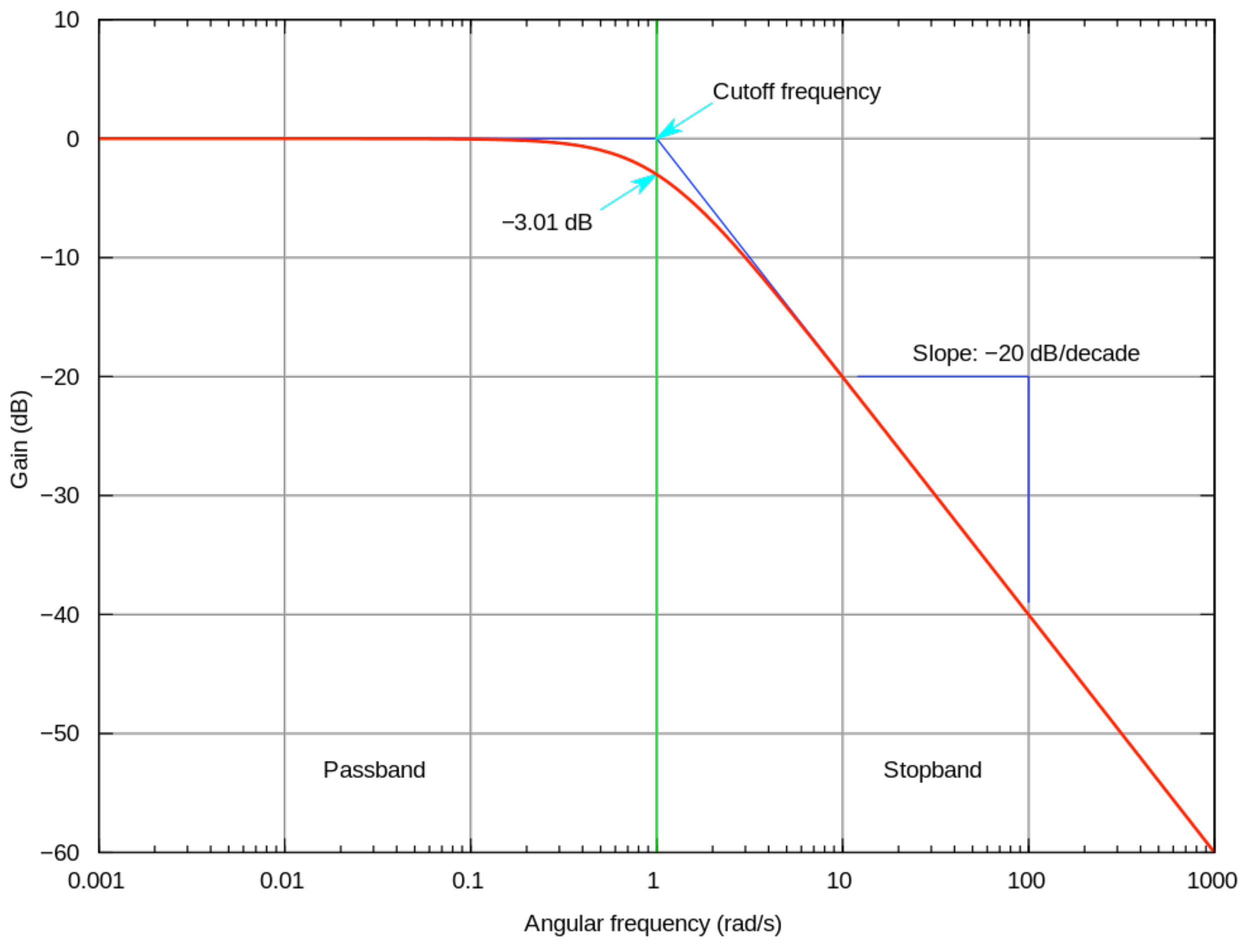
Two (tiny) plates separated by a non-conductive material (dielectric)

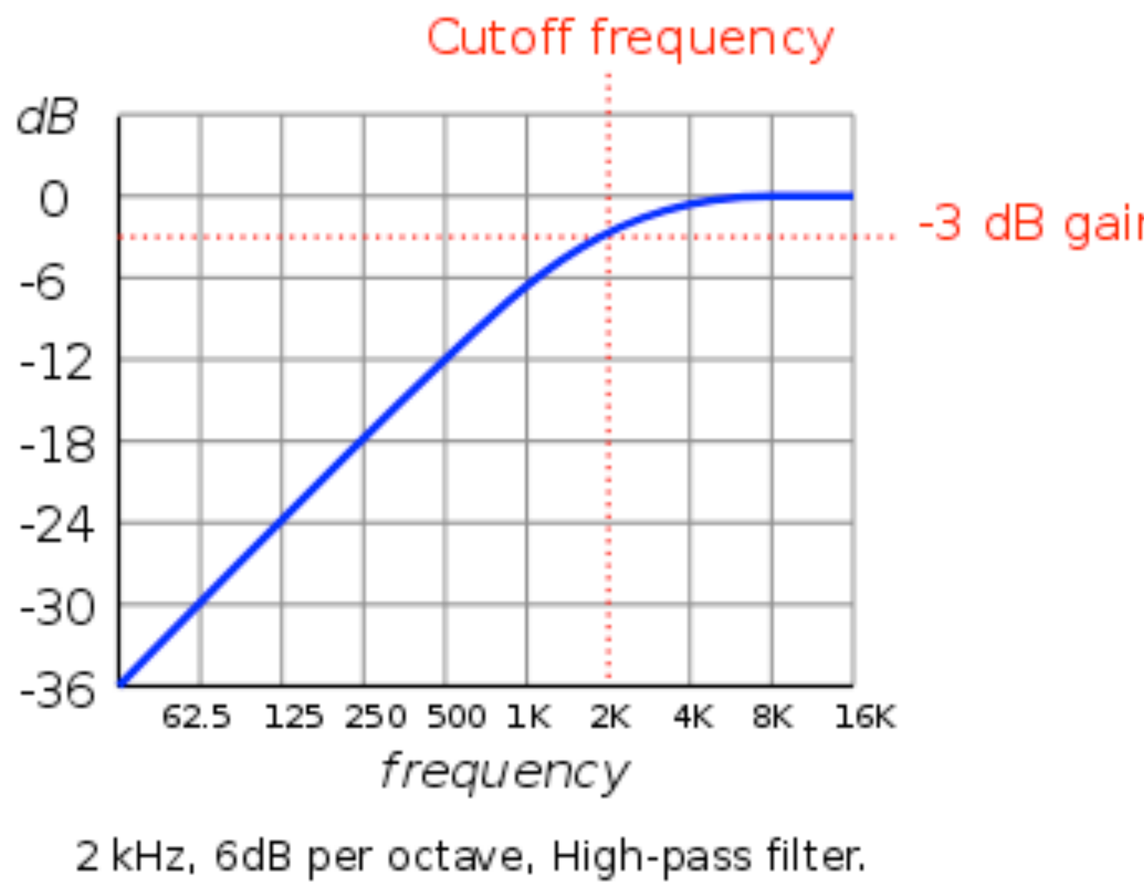


Frequency-selective impedance



Filters

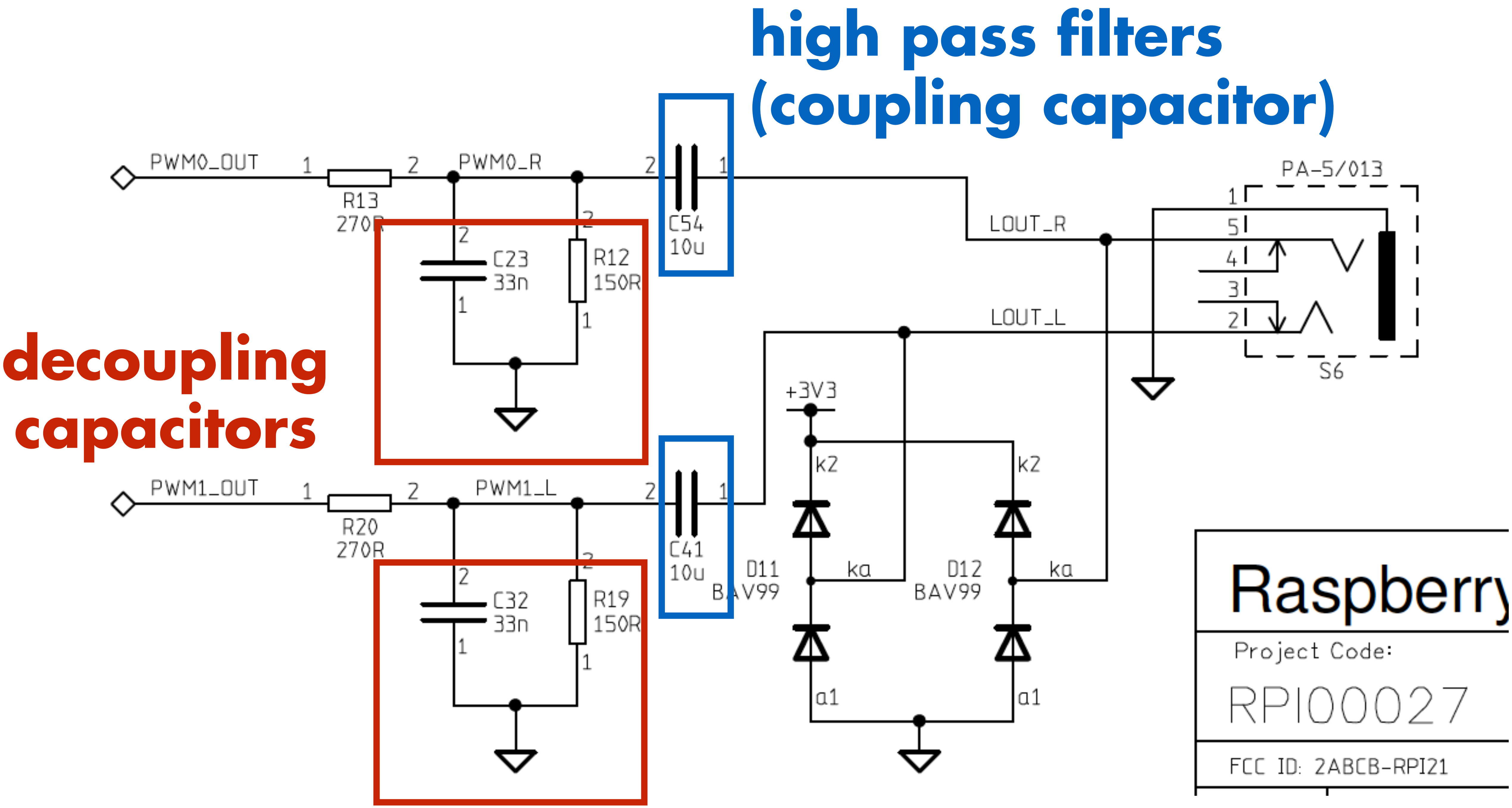




low pass

high pass

Raspberry Pi Audio Circuit



Hardware PWM Support

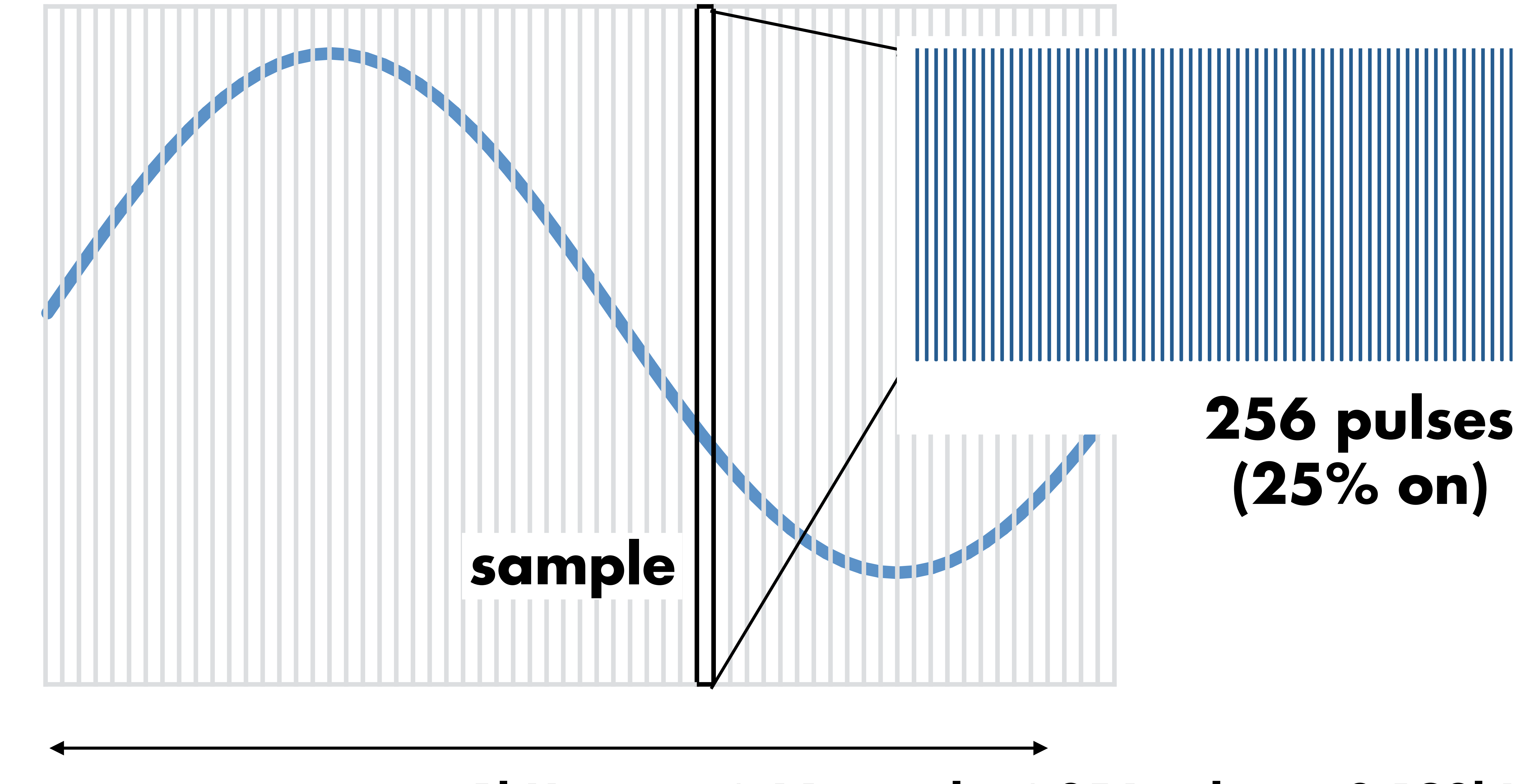
Start with a 19.2MHz clock, divide it to specify the time slots of on/off

E.g., divider of 2.375 = 8,192kHz

Divide wave into steps (e.g., 64)

Divide each step into train of (e.g., 256) pulses: tell hardware how many pulses should be high

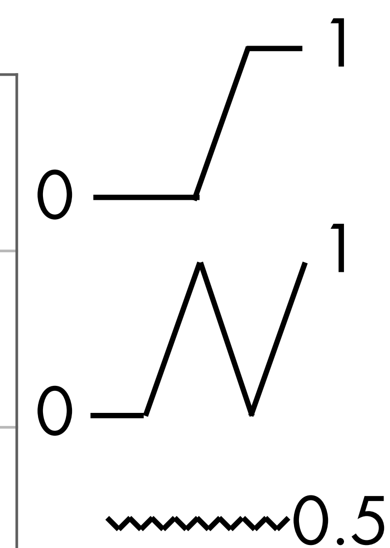
Example: Sine



1kHz wave * 64 samples * 256 pulses = 8,192kHz

PWM Clocking of Pulses

bad	0	0	0	0	0	0	0	0	+V	+V	+V	+V	+V	+V	+V	+V
fair	0	0	0	0	+V	+V	+V	+V	0	0	0	0	+V	+V	+V	+V
good	0	+V	0	+V	0	+V	0	+V	0	+V	0	+V	0	+V	0	+V



step

What if we want real music?

MIDI

*Actually, this is kind of fake music

MIDI: Musical Instrument Digital Interface

Simple interface to control musical instruments

Emerged from electronic music and instruments in 1970s

First version described in Keyboard magazine in 1982

A bit of “music”

MIDI

31.25 kbps 8-N-1 serial protocol

Commands are 1 byte, with variable parameters (c=channel, k=key, v=velocity, l=low bits, m=high bits)

Command	Code	Param	Param
Note on	`1001cccc`	`0kkkkkkk`	`0vvvvvvv`
Note off	`1000cccc`	`0kkkkkkk`	`0vvvvvvv`
Pitch bender	`1110cccc`	`0lllllll`	`0mmmmmmm`

UART (2+ pins)

Bidirectional data transfer, no clock line — "asynchronous".

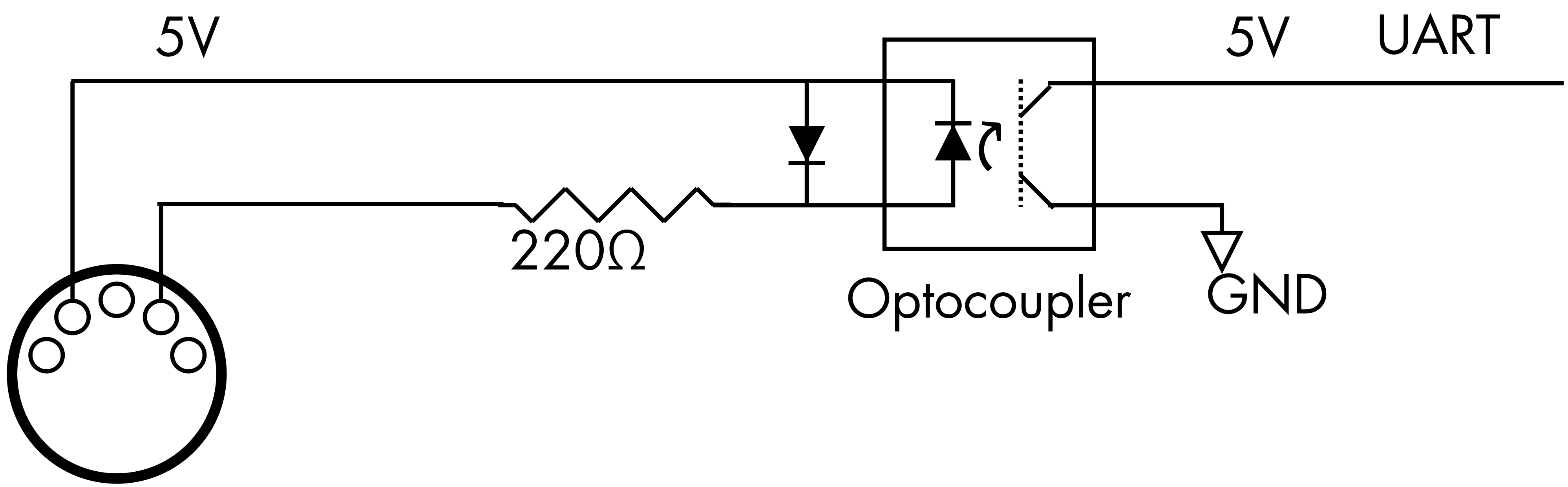
Additional pins for flow control ("I'm ready to send"), old telephony mechanisms.

Start bit, (5 to 9) data bits, (0 or 1) parity bit, (1 or 2) stop bit. 8-N-1:

start	data	data	data	data	data	data	data	data	parity	stop	stop
0	d_1	d_1	d_1	d_1	d_1	d_1	d_1	d_1		1	1

MIDI Circuit

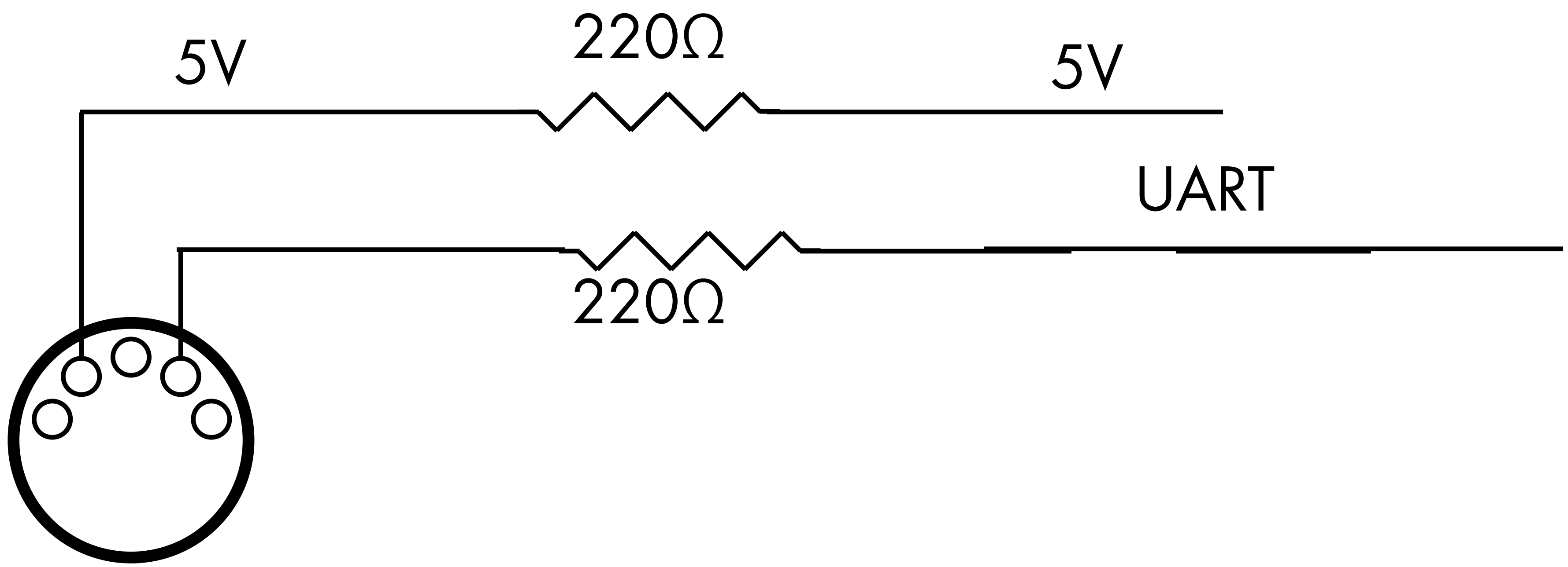
0 is high, 1 is low!



Optocoupler completely isolates circuits electrically: no noise in instrument

MIDI Hack!

Volca Beats (looking into midi player), reversed for Pi



`code/midi`

Raspberry Pi hooked up to KORG volta keys on GPIO pin 25.

UART timing

Inversion