Where are We Going?

Processor and memory architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

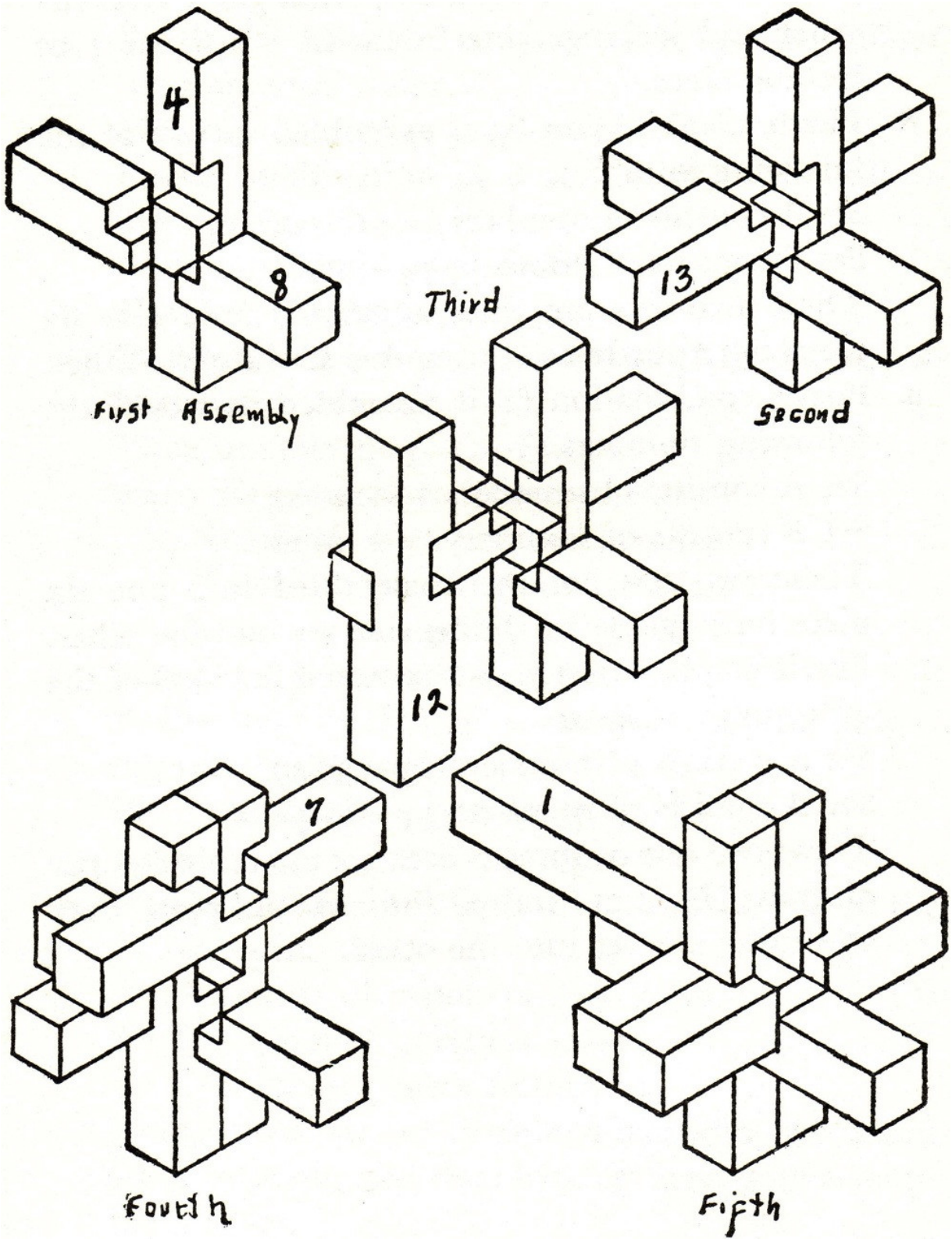
From C to assembly language

Function calls and stack frames

Serial communication and strings

Modules and libraries: Building and linking

Memory management: Memory map & heap


4
8
First Assembly
13
Second
Third
12
7
Fourth
1
Fifth

gpio
timer
uart
printf
malloc
keyboard
fb
gl
console
shell

Good Modules

Meaningful decomposition of the system into parts (modules)

Interfaces and implementations

- **Provide easy-to-use interface to users**
- **Hide obscure details of implementation**

Tested independently with unit tests

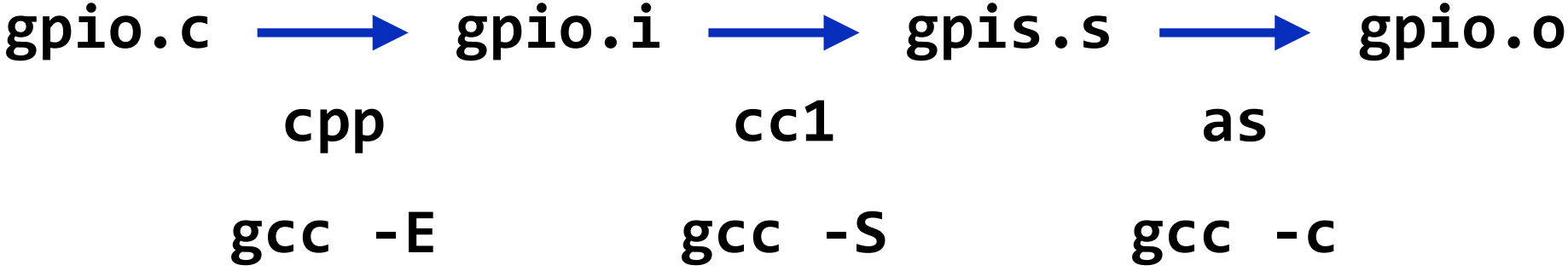
Obi-wan, how should I design my modules?

"The Build"

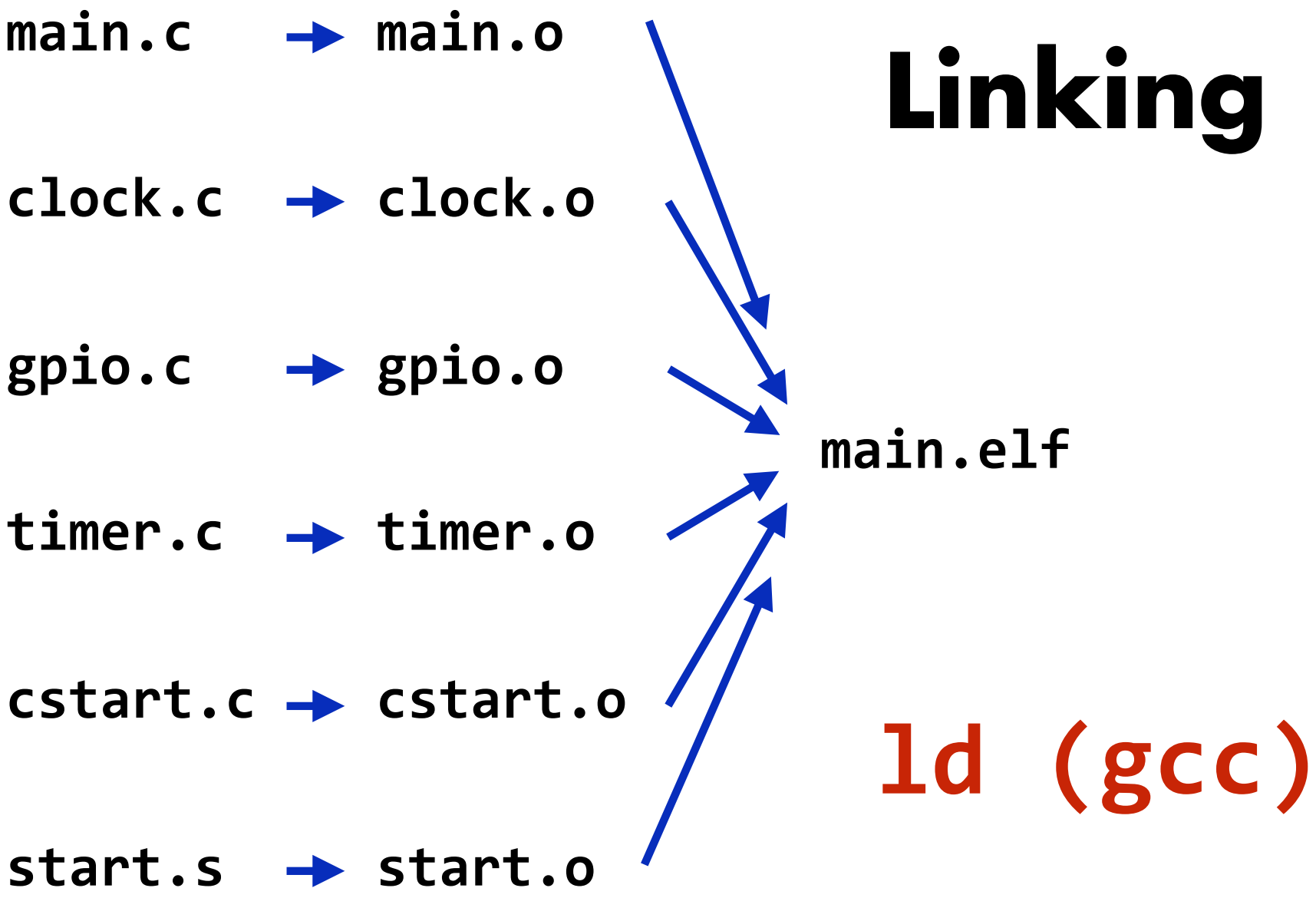
NASA Command Center during SpaceX Mission

cs107e Command Center: `Makefile`

gcc is all powerful



gcc –save-temps


main.c
main.o
clock.c
clock.o
gpio.c
gpio.o
timer.c
timer.o
cstart.c
cstart.o
start.s
start.o
main.elf
Linking
ld (gcc)

Common Errors

1. Symbol undefined
2. Symbol multiply defined

Symbols

Single global name space

- **need** `gpio_` **prefix to distinguish names**
- **e.g.** `gpio_init` **versus** `timer_init`

Local variables in functions are not symbols

Defined vs. undefined (`extern`) **symbols**

Definitions: global vs local (`static`)

- **by default symbols are local in** `.s` **files**
- **by default symbols are global in** `.c` **files**

Symbol Resolution

Set of defined symbols D

Set of undefined symbols U

Moving left to right, for each .o file, the linker updates U and D and proceeds to next input file.

Problems

- **If two files try to define the same symbol, an error is reported***
- **After all the input arguments are processed, if U is found to be not empty, the linker prints an error report and terminates.**

Libraries

An archive `.a` is just a collection of `.o` files.

The linker scans the library for any `.o` files that contain definitions of undefined symbols. If a file in the library contains an undefined symbol, the whole file and all its functions are linked in.

Adding the `.o` file from the library may result in more undefined symbols; the linker searches for the definition of these symbols in the library and includes the relevant files; this search is repeated until no more definitions of undefined symbols are found.

Questions?

Suppose you add more functions to the clock interface (e.g. `clock_start()`, `clock_tick()`, `clock_end()`) what source files would need to be modified? rebuilt?

Can you think of a way to force linking if you change `OBJECTS` in a `Makefile`?

What happens if you link with

```
ld … -lpi gpio.o?
```

When to Rebuild?

Change to implementation (`clock.c`)?

- **Must recompile implementation (`clock.o`)**

Change to interface (`clock.h`)?

- **Should (must) recompile clients of the interface (`main.c`)**
- **Add recipe that `main.o` depends on `clock.h`**

Change to `Makefile`

- **Adding a file to `OBJECTS` may require rebuilding executable `main.elf`**
- **Modify recipe for `main.elf` to depend on `Makefile`**
- **BEWARE: This is typical of a hidden dependency**

Combining Multiple Modules (`.o`) into a Single Executable (`.elf`)

`memmap`

```
// memmap
MEMORY
{
    ram : ORIGIN = 0x8000,
          LENGTH = 0x8000000
}
.text : {
    start.o (.text)
    *(.text*)
} > ram

// Why must start.o go first?
```

_start must be located at #0x8000!!

```
% arm-none-eabi-nm -n main.elf
00008000 T _start
00008008 t hang
0000800c T main
00008020 T timer_init
00008024 T timer_get_time
0000802c T delay_us
00008038 T delay
0000806c T gpio_init
00008070 T gpio_set_function
00008074 T gpio_get_function
00008078 T gpio_write
0000807c T gpio_read
00008080 T clock_init
00008084 T clock_run
00008088 T _cstart
000080d8 T __bss_end__
000080d8 T __bss_start__
```

```
# size reports the size of the text
% arm-none-eabi-size main.elf
   text    data     bss     dec     hex filename
    216       0       0     216      d8 main.elf

% arm-none-eabi-size *.o
   text    data     bss     dec     hex filename
      8       0       0       8       8 clock.o
     80       0       0      80      50 cstart.o
     20       0       0      20      14 gpio.o
     20       0       0      20      14 main.o
     12       0       0      12       c start.o
     76       0       0      76      4c timer.o

# Note that the sum of the sizes of the .o's
# is equal to the size of the main.exe
```

Relocation

```
// start.s

.globl _start
start:
    mov sp, #0x8000
    mov fp, #0
    bl _cstart
hang:
    b hang
```

```
// Disassembly of start.o (start.list)

0000000 <_start>:
   0:   mov sp, #0x8000
   4:   mov fp, #0
   8:   bl  0 <_cstart>

0000000c <hang>:
   c:   b   c <hang>

// Note: the address of _cstart is 0
// Why?
//   _start doesn't know where c_start is!

// Note it does know the address of hang
```

```
// Disassembly of main.elf.list

00008000 <_start>:
    80ΩΩ: mov     sp, #134217728  ;
0x8000000
    8004: bl      8088 <_cstart>

00008008 <hang>:
    8008: b       8008 <hang>

00008088 <_cstart>:
    8088:   push    {r3, lr}

// Note: the address of _cstart is #8088
// Now _start knows where _cstart is!
```

data/

```
// uninitialized global and static
int i;
static int j;

// initialized global and static
int k = 1;
static int l = 2;

// initialized global and static const
const int m = 3;
static const int n = 4;
```

```
% arm-none-eabi-nm -S tricky.o
00000004 00000004 C i
00000000 00000004 b j
00000000 00000004 D k
00000004 00000004 d l
00000000 00000004 R m
                  U p
00000000 00000048 T tricky

# The global uninitialized variable i
# is in common (C).

# The static const variable n
# has been optimized out.
```

Guide to Symbols

T/t - text

D/d - read-write data

R/r - read-only data

B/b - bss (*Block Started by Symbol*)

C - common (instead of B)

lower-case letter means `static`

Data Symbols

Types

- **global vs static**
- **read-only data vs data**
- **initialized vs uninitialized data**
- **common (shared data)**

```
.text : {
    start.o (.text)
    *(.text*)
} > ram
.data : { *(.data*) } > ram
.rodata : { *(.rodata*) } > ram
__bss_start__ = .;
.bss : {
    *(.bss*)
    *(COMMON)
} > ram
. = ALIGN(8);
__bss_end__ = .;
```

Sections

Instructions go in `.text`

Data goes in `.data`

`const` data (read-only) goes in `.rodata`

Uninitialized data goes in `.bss`

+ other information about the program

- **symbols, relocation, debugging, ...**

Builds

Automate the build! Manual builds are error prone

Needs to be fast and reliable

- **Fast means compile modules only when necessary**
- **Reliably means keeping track of dependencies between files**

Separate system into small modules with minimal dependencies

Ensure `Makefile` contains all dependencies