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

Almost Half-Way

How are things going?

- From the staff perspective: things are going fine, and we know that the material is challenging and the assignments are time-consuming. If you feel behind: you probably aren't!
- We know you are working hard! We know this can be frustrating, but we also hope you are learning a ton, and also enjoying it!

gpio
timer
uart
printf
malloc
keyboard
fb
gl
console
shell

Good Modules

Decompose a system into smaller parts (modules)

- Interface: what the module does
- Implementation: how it does it

A good interface

- An easy-to-understand abstraction that simplifies code
- Can be implemented easily and in different ways

Tested independently with unit tests

Example: printf

```
int printf(const char* format, …);
```

printf() does a lot of things

- Parses a format string
- Converts arguments into strings
- Concatenates format string and arguments into a longer string
- Outputs string to terminal/serial port

Example: printf

```
int printf(const char* format, …);
```

printf() does a lot of things

1. Outputs string to terminal/serial port
2. Concatenates format string and arguments into a longer string
3. Parses a format string
4. Converts arguments into strings

An example decomposition, recommended in the assignment, that breaks the problem into smaller, simpler parts.

Example: printf

```
int printf(const char* format, …);
```

printf() does a lot of things

1. `printf()`
2. `snprintf()`
3.
4. `sign_to_base()`

An example decomposition, recommended in the assignment, that breaks the problem into smaller, simpler parts.

Linking

Your program uses multiple modules, each with its own abstraction and logic

Some modules call other modules: you need module A to be able to invoke a function in module B

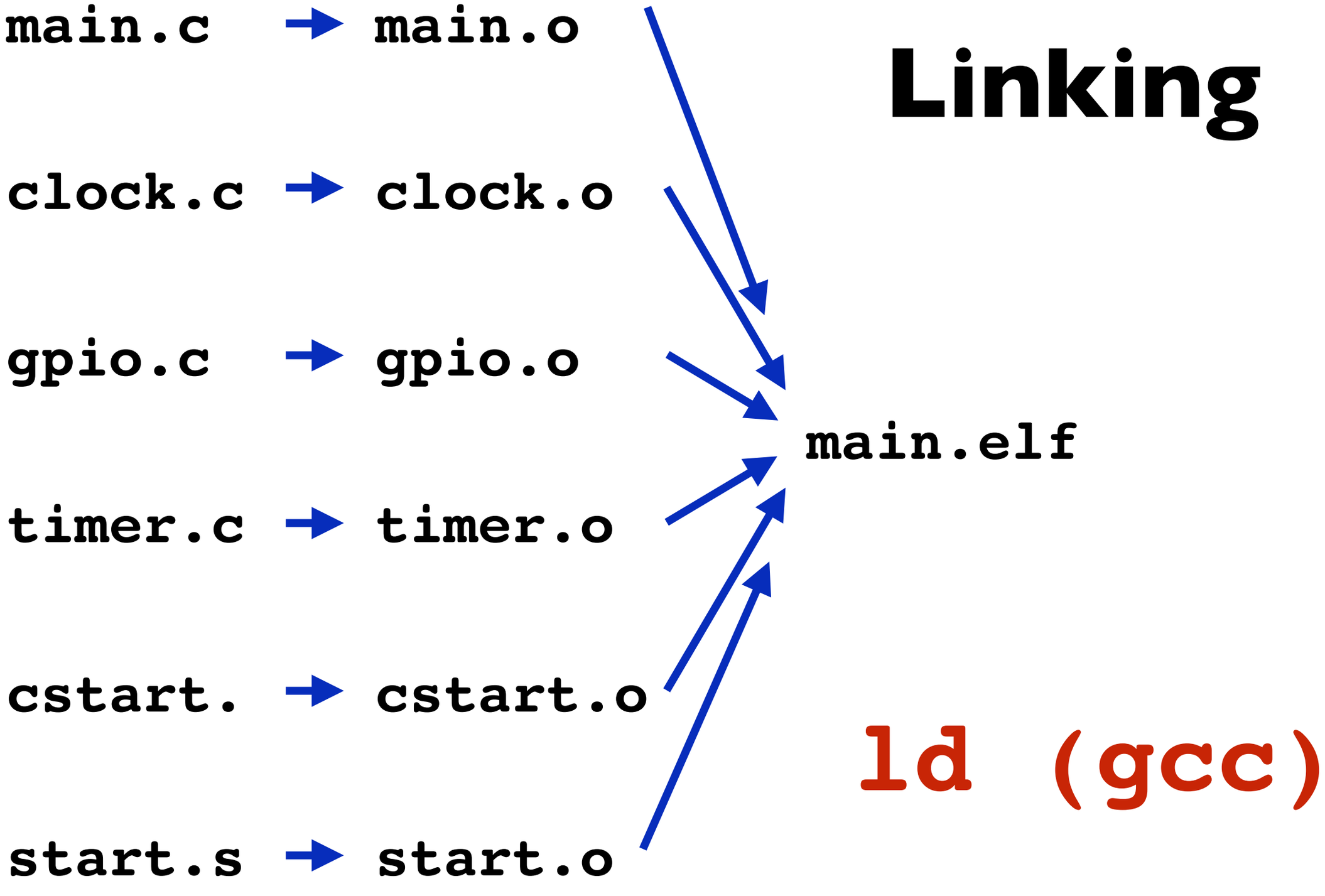
Your tools *link* them together, somehow combining them into a single block of binary code

"The Build"

gcc is all powerful

gcc -E **gcc -S** **gcc -c**

gcc –save-temps


main.c
main.o
clock.c
clock.o
gpio.c
gpio.o
timer.c
timer.o
cstart.
cstart.o
start.s
start.o
Linking
main.elf
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

Question

C doesn't support polymorphic functions.

You can't have both

- `itoa`(char *buf, int bufsize, short val);
- `itoa`(char *buf, int bufsize, int val);
- There can be only one symbol `itoa` in object files

C++ *does* support polymorphic functions

- `ostream& operator<< (short val);`
- `ostream& operator<< (unsigned short val);`
- How does this work?

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?

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!!

Magic constant that's part of Raspberry Pi boot sequence.

```
$ arm-none-eabi-nm -n clock.elf
00008000 T _start
0000800c t hang
00008010 T square
0000801c T blink
00008070 T main
0000808c T timer_init
00008090 T timer_get_ticks
00008098 T timer_delay_us
000080a4 T timer_delay_ms
000080c0 T timer_delay
000080e0 T gpio_init
000080e4 T gpio_set_function
000080e8 T gpio_get_function
000080f0 T gpio_set_input
000080f4 T gpio_set_output
000080f8 T gpio_write
000080fc T gpio_read
00008104 T _cstart
00008154 T __bss_start__
00008158 T __bss_end__
```

```
# 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, #0x8000000
    mov fp, #0
    bl _cstart
hang:
    b hang
```

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

0000000 <_start>:
   0:   mov sp, #0x8000000
   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 clock.elf.list

00008000 <_start>:
    8000: 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;
int l = 0;
static int m = 2;
```

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

```
% arm-none-eabi-nm -S tricky.o
00000004 00000004 C i
00000000 b j
00000000 00000004 D k
00000004 00000004 B l
00000004 00000004 d m
00000000 00000004 R n
00000004 00000004 r o
00000000 000000e4 T tricky

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

# If you compile with -Og, some variables
# get optimized out…which ones?
```

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)

Sections

Instructions go in `.text`

Data goes in `.data`

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

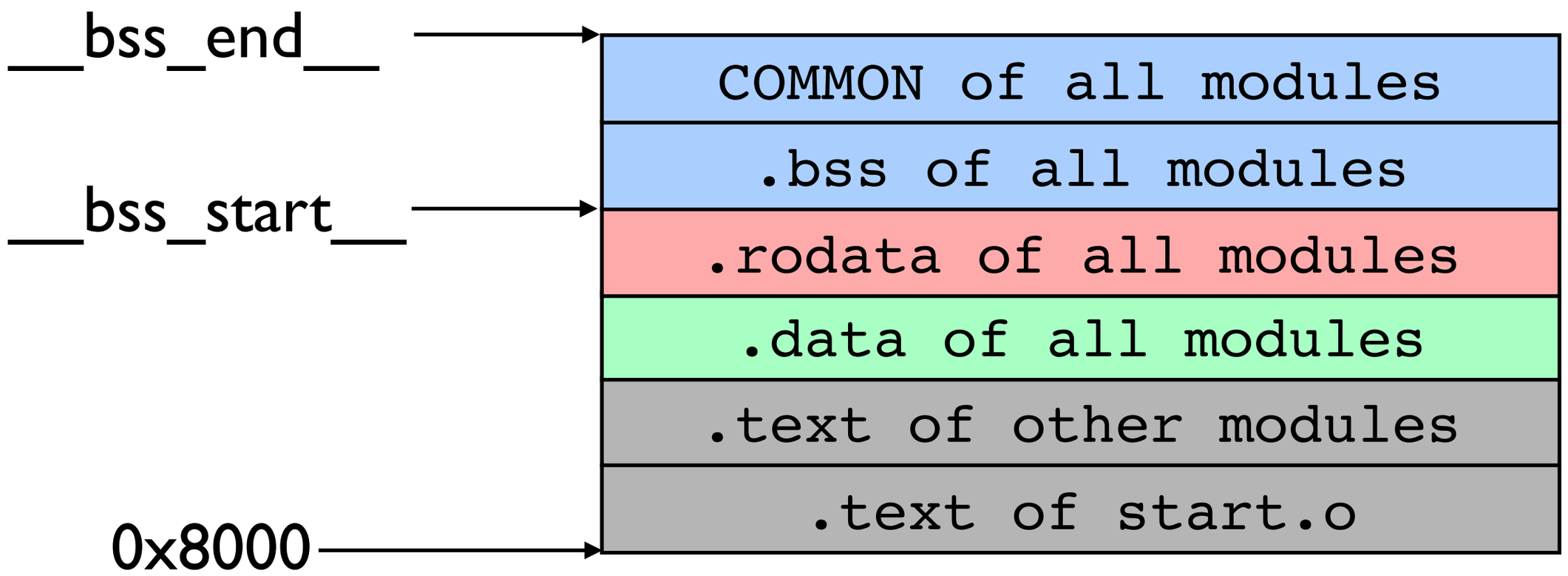
Uninitialized data goes in `.bss`

+ other information about the program

- symbols, relocation, debugging, …

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

```
MEMORY {
  ram : ORIGIN = 0x8000,
        LENGTH = 0x8000000
}
.text : {
        start.o (.text)
        *(.text*)
    } > ram
    .data : { *(.data*) } > ram
    .rodata : { *(.rodata*) } > ram
    __bss_start__ = .;
    .bss : {
        *(.bss*)
        *(COMMON)
    } > ram
    . = ALIGN(8);
    __bss_end__ = .;
```



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

Summary

Decomposing software into modules is a critical skill

- Like organizing an essay into sections, paragraphs, sentences

The linker combines modules into a larger binary

- Resolves undefined symbols to modules that define them
- Lays out data and code
- Relocation when needed

Makefiles designed to only compile modules that need recompilation

- If F changes, recompile everything that depends on F