Where are We Going?

Processor and memory architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

From C to assembly language

Functions and stack frames

Serial communication and strings

Modules and libraries: Linking

Memory Map: Loading, starting, and the heap

gpio
timer
uart
printf
malloc
keyboard
mailbox
fb
gl
console
monitor

blink/

Why Modules?

Logical decomposition of the system into parts

Interfaces and implementations

- **Clean and easy-to-use abstractions**
- **Hide obscure details**
- **Isolate consumers from providers**

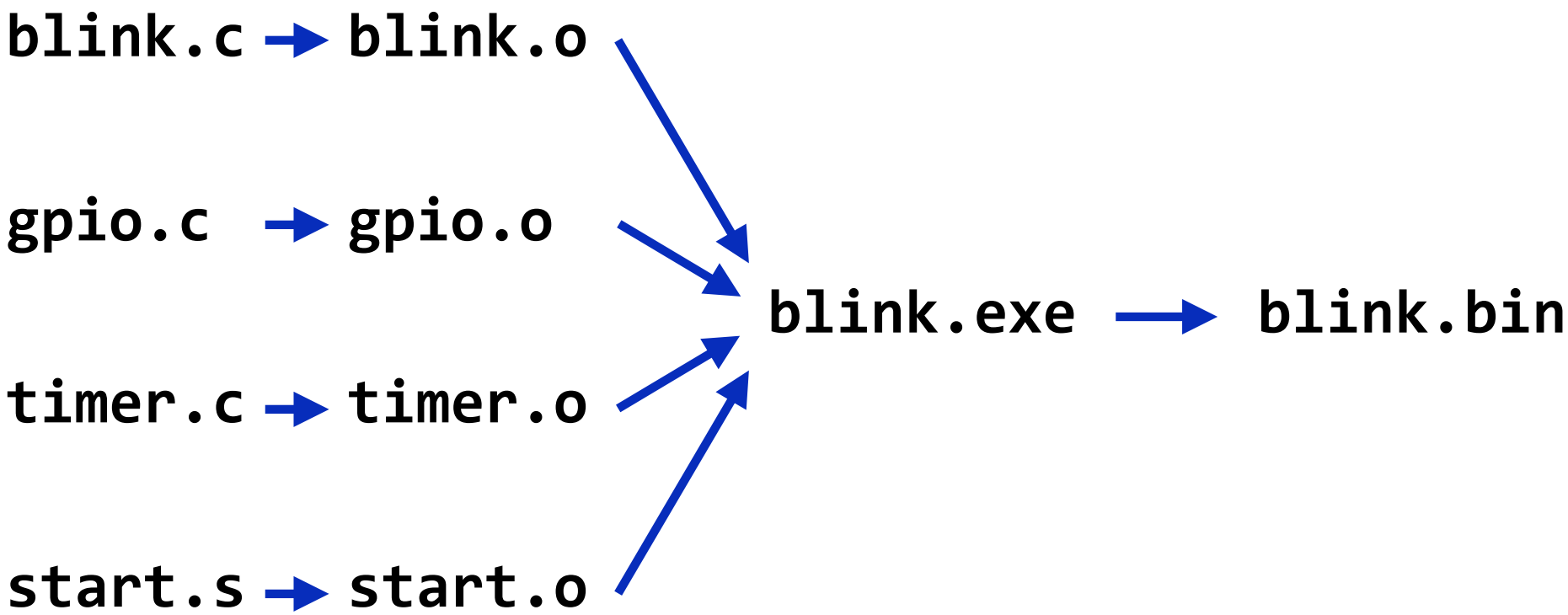
Tested independently with unit tests

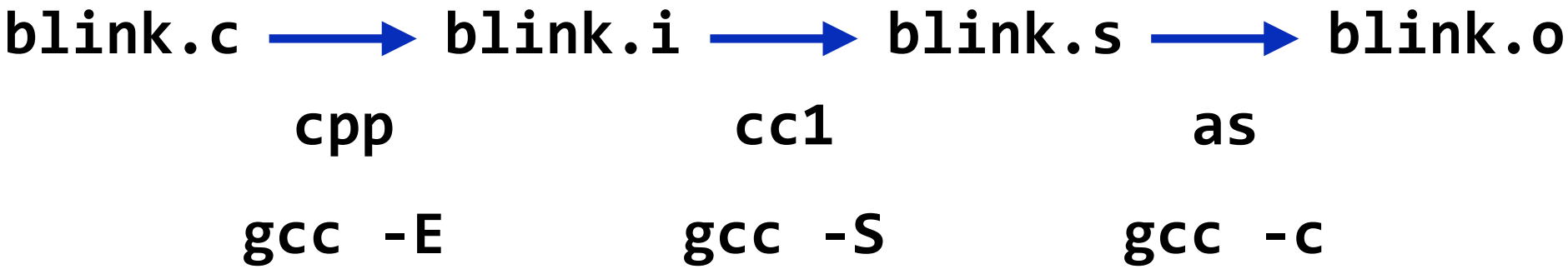
Obi-wan, how can I design powerful modules?

blink/Makefile

How to Build Parts?

```
NAME = blink
OBJECTS = gpio.o timer.o start.o

$(NAME).exe: $(NAME).o $(OBJECTS)
    arm-none-eabi-ld $(LDFLAGS)  $^ -o $@

%.o: %.c
    arm-none-eabi-gcc $(CFLAGS) -c $< -o $@

%.o: %.s
    cpp -P $< | arm-none-eabi-as $(ASFLAGS) -o $@

%.list: %.o
    arm-none-eabi-objdump -d $< > $@

%.bin: %.exe
    arm-none-eabi-objcopy $< -O binary $@
```


blink.c
blink.o
gpio.c
gpio.o
timer.c
timer.o
start.s
start.o
blink.exe
blink.bin



gcc --savetemps

When to Build Parts?

Build target if it does not exist

Rebuild target if any of the dependencies have changed (are newer than the target)

WARNING: only considers explicit dependencies; there may be other dependencies

Linking

`blink/`

Symbols

Definitions: global vs local (`static`)

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

Defined vs. undefined (`extern`)

Single global name space

- **need `gpio_` prefix to prevent collisions**

Local variables in functions are not symbols

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 (unless it is a COMMON).**
- **After all the input arguments are processed, if U is found to be not empty, the linker prints an error report and terminates.**


Intermission

Questions?

Suppose you edit your Makefile, what kinds of changes would require rebuilding the `.o`'s

Suppose you change the interface in a header file, what do you need to rebuild?

When to Build?

change implementation (`gpio.c`)?

- **recompile implementation (`gpio.c`)**

change interface (`gpio.h/gpio.c`)?

- **must recompile clients of the interface (`blink.c`)**
- **`blink.c` depends on `gpio.h`**

data/

Types of Symbols

Text

- `T/t` **- instructions / code (executable data)**

Data

- `D/d` **- read-write data**
- `R/r` **- read-only data**
- `B/b` **- bss (uninitialized block storage)**
- `C` **- common**

Lower-case means local (`static`) / upper-case global

Relocation

```
// start.s

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

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

00000000 <_start>:
   0:   mov sp, #32768  ; 0x8000
   4:   bl  0 <main>

00000008 <hang>:
   8:   b   8 <hang>

// Note: the address of main is 0
// Why? It doesn't know where main is!
```

```
// Disassembly of blink.exe.list

00008000 <_start>:
    8000:   mov sp, #32768  ; 0x8000
    8004:   bl  800c <main>

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

0000800c <main>:
    800c:   push    {r3, lr}

// Note: the address of main is #800c
// Now it knows where main is!
```

Libraries

`libpi/`

Symbols from in 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, it is included in the linker file list

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

Development Tasks

Managing changes to the source

Compiling pieces individually

Combining ("linking") the pieces together into a single executable program

Building the systems quickly and reliably

- **Quickly means incrementally**
- **Reliably means rebuilding modules when necessary**

Your Tools are "*Special*"

Programs
That Manipulate
Programs