PreProcessor

File.c → [preprocessor] → File.i

* Text replacement.
* delete comments
* Works seperately on each file
* بينفذ العمليات اللي بتبدأ بـ #

---

* # define — macro template expansion لو كتبنا مثلا

#define Max 5 ⇒ هيبدل أي Max في الكود بـ 5

Portability, reusability, readability ← طب إحنا ليه نستخدم #define في حالة زي اللي فوق؟ ← عشان الـ

- magic number ⇒ أي رقم بتكتبه لوحده في الكود — int x=5;

- ممكن أعمل أكتر من سطر عن طريق concatination operator '\'

define U8 unsigned char — هتشتغل عادي ومفيهاش مشكلة

define U8P unsigned char*

لو روحت في الـ main: U8P P1, P2; — بعدها ⇒ unsigned char* P1, P2;
دي بس اللي بوينتر ↓ P1 — ↓ P2 char

فيه حل للمشكلة دي keyword

→ typedef unsigned char* U8P; — بتعمل نوع جديد

By compiler — U8P P1, P2; ⇒ الإتنين Pointers

* ميتفعش نستخدمه جوه switch

* macro like Function

#define SQR(x) x*x

في الـ main: int x=5; printf("%d", SQR(x+2)); — هتطلع فعليا بـ 17 ليه؟

لأن بـ #define بت replace بعدين execute ففي المثال ده كانت 17 = x+2*x+2

عشان نحل مشكلة زي دي نخلي الـ expansion في أقواس #define SQR(x) ((x)*(x))

⇒ ternary operator ? — x>y ? x : y (if / then / else) — مهم في #define

	Function (Function Call 2 line, 100 line)	Macro like Function (macro 100 line)
code size	كم line؟ لو call مرة واحدة 102، لو مرتين 104، لو 4 108	لو مرة واحدة 100، لو مرتين 200، لو 4 400
Time	لو الـ body 100 cycle، والـ call والـ return 4 cycle — 416	هياخد وقت 400
	مفترضين 2 lines	مفترض 2 lines
code size	Call 4 times — 10 lines	8 lines
Time	24 cycle	8 cycles
	Check argument	doesn't check argument
	لو فيه error فسهل أوصله	الـ error صعب أوصله

* Header file

* لو أنا في الـ main وعملت include لـ header file والفايل ده مش موجود هيجيلي error
في مرحلة الـ Preprocessor ← no such file or directory

#if #elif #else #endif مش بتحتاج أقواس زي اللي بين ايديهم
شبه if statement بس بتمسح الكود اللي مش عايزه يعني الـ false بتمسحه
لازم الشرط بتاعي يكون حاجة تتفهم في الـ Preprocessor مثلاً #define

#ifdef لو الكلمة موجودة في #define
#ifndef لو الكلمة مش موجودة في #define
#warning "message" أثناء الـ build time أقدر أستخدمها وسيب أي حاجة بعدها مع #if
#error error message → stops compilation
* File guard
بنعمله في الـ header عشان لو أنا مش واخد بالي عملت include للفايل أكتر من مرة الـ guard بيخليه يعتبره مرة واحدة بس، كله هيتمسح ويفضل واحد بس

```
#ifndef FILE_H
#define FILE_H
:
#endif
```

#Pragma : used to turn on or off some features , compiler dependant
ex: #pragma once تشتغل كـ file guard
↳ Add memory section فيها attributes

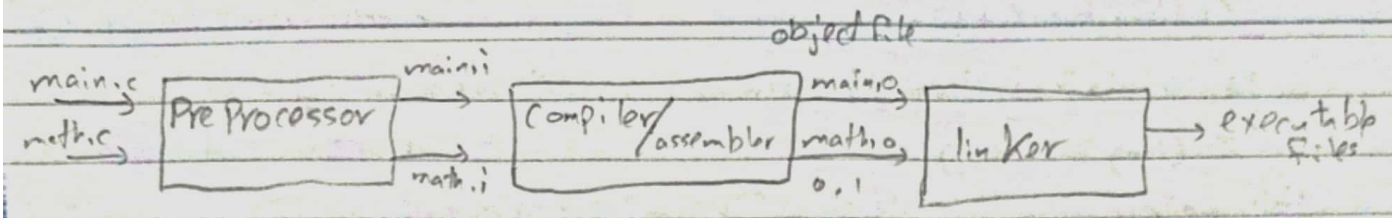
#include "Path"

- Absolute Path — PC الى Path
- Relative Path → Current directory
  - #include "New/New2/Temp.h"
  - go back #include "../lib.h"

أحسن متستخدمش الـ Absolute عشان لو عايز أنقل على PC تاني هتلاقي مشكل يحل مشكلة

Build process

(tool chain)



* pre processor
  - text replacement * define * include
  - works separately on each file.
* Compiler/assembler.-
  - translate C ⟶ machine language (0,1)
  - works separately each file.
  - doesn't allocate in memory
* linker
  - links all Files together
  - allocate in memory.

- الكومبايلر بيترجم الى جروب الـ file - اللينكر بيترجم الـ files مع بعض.

- compiler is responsible for arranging elements inside the groups,
  linker ⟹ arrange the groups together

G_1	G_2	G_3
1	11	21
2	12	22
3	13	1
⋮	⋮	⋮
10	20	30

* Linker error: ld error happens while combining the files together
  ex:- more than one function with the same name
    - ⟶ same file ⟹ compiler error
    - ⟶ different files ⟹ linker error
  ex:- called a function in main, but there is no function with that name in any file ⟹ linker error

(...) → anything else كأن syntax #define فيه في

↳ value store var → (__VA_ARGS__)

ex. #define Eyad(...) Printf(__VA_ARGS__)

Eyad("Hello"); لوفي ال main كتبت

هتتنقل ك Printf على

ex. #define Fun(a,...) Printf(__VA_ARGS__, a)

في ال main Fun(x, "x=%d"); ≡ printf("x=%d", x);

⇒ Predefined macros:- Built_in ⇒ __DATE__

ممكن نستعملهم في ال debugging

* Stringfication:- # بحول ل string (strinize operator)

define Printf(X) Printf(#X) → بتحوطها ال " "

* Concatination:- ## (Token_Pasting operator) هنستخدمها PORT driver

#define conc(x,y) x##y

في ال main int x = conc(3,8); ≡ int x = 38;

.c → Tool chain → executable file

preprocessor + compiler + Assembler + linker + dedugger + libiraries

+ # helper programer → binary utilities → obj copy, obj dump, readelf (analysis)

⇒ Types: - Native: compile and run at the same place (PC) مثلا code blocks زي

- cross: build at PC مثلا, run في MCU (avr) مثلا

⇒ Systems: - Build system: source code of toolchain GCC native

- Host system: ↓ GSS cross compiler

- Target system: ↓ MCU

compile بعمل فيه file تطلع

* Libraries → static : source code not available

Dynamic: open source

To get static library: .c .h → .o → (.a) → (.a, .h) library

cmd في ال — gcc -c name.c -o name.o — طلعت ال object file

ar rcs lib_name.a name.o — طلعت ال library

أنا لو عوزت معايا ال .a , .h

static library — gcc main.c lib_name.a -o main.exe

محتاج أعمل compile عشان — طلعت executable file

- To add another object file to lib — ar r lib_name.a file.o
- To delete one from lib — ar d lib_name.a file.o
- To view all object files in lib — ar t lib_name.a
- To extract object files — ar x lib_name.a

* ARM Cross tool chain (GNU GCC , arm cc)
  - GNU GCC ↓ Free, open source
  - arm cc ↓ licensed

= arm-none-eabi-gcc ⇒ Baremetal app → run على arm لوحده

= arm-Linux-gnueabi ⇒ (os app)

Build process

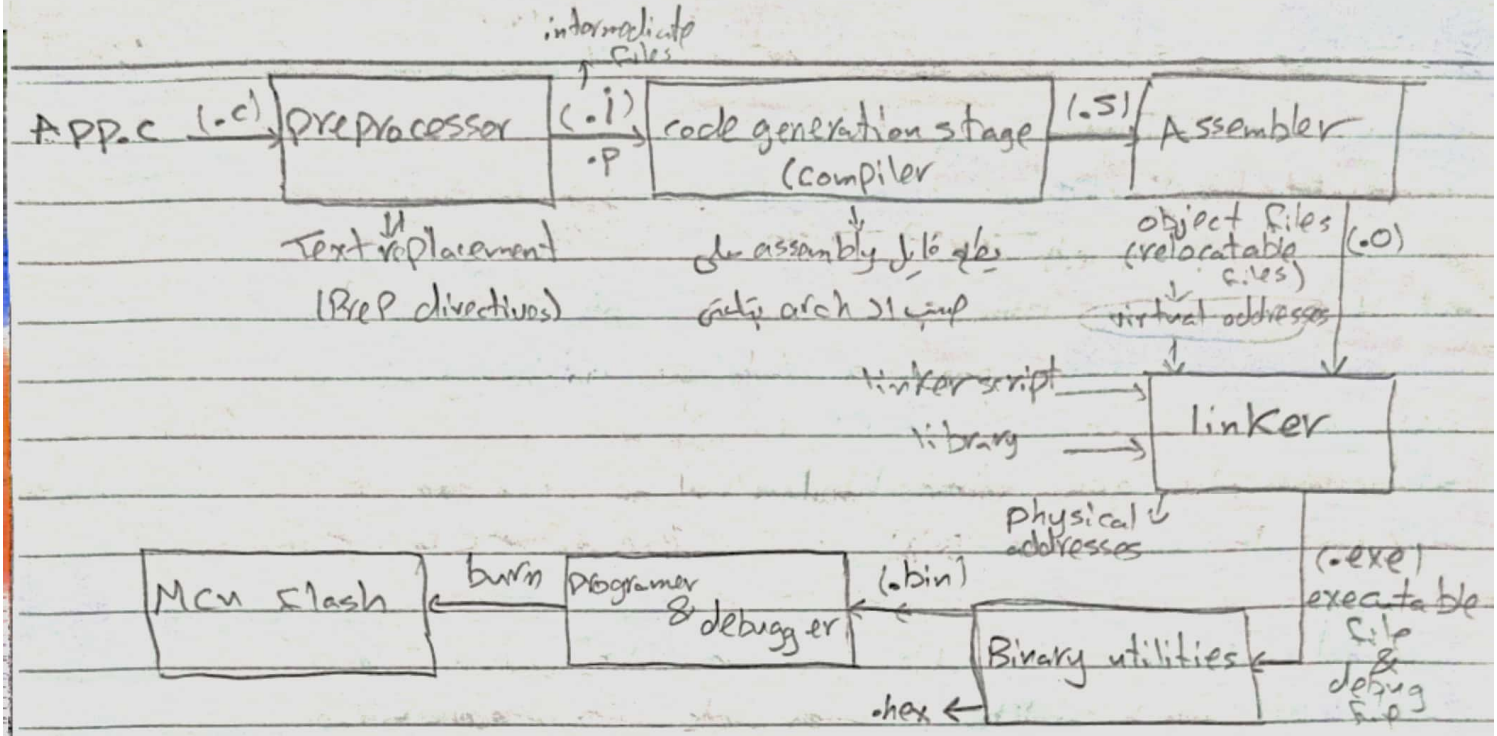
أول مرحلة

App.c —.c→ [preprocessor] → .i / .p (intermediate files)

⇓

- Text Replacement (Preprocessor directives)
- عدد فايلات .i = عدد فايلات .c

Build Process

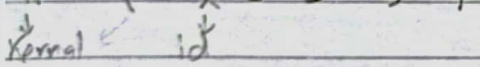


* Compiler:- (Front end - middle - backend stage)

① Frontend stage: - tokenizer <-> → tokens (keywords, operators, Identifiers)
lexical analysis - syntax analysis
- check syntax errors.
- o/p → Parse tree → IR program
semantic table

if (x == 5)

Parsing tree



* IR (Intermediate Representation): the data structure or code used internally by a compiler to represent source code

② Middle stage: - Semantic analysis: logic check
- optimization: Fix code (code size ↓, RAM consumption ↓, Power consumption ↓, execution time ↓)

How? Multilevel process: - remove dead code (unreachable)
- inline expansion of function (func → inline func)
- Register allocation (GPR)
- loop unrolling: (execution time ↓, code size ↑)

→ عشان ميحجزش غير الحجم

At compile-time, the compiler knows the size of an array. the compiler will decide where in memory should the array go, then at run-time, the memory for the array will be allocated

③ Backend stage:

- Code generation: convert into assembly
- → Memory Allocation:

لسه منا آكرتهاش ← memory sections

code → .txt

initialized global, static → .data

constants → .rodata

uninitialized static, unintialized global → .bss

auto, local → stack

*Notes

symbol	Address
✓	✓
	virtual not physical

- في مرحلة ال compiler ⇐ symbol table
  - ↳ variable, func names
  - ↳ private (static), global

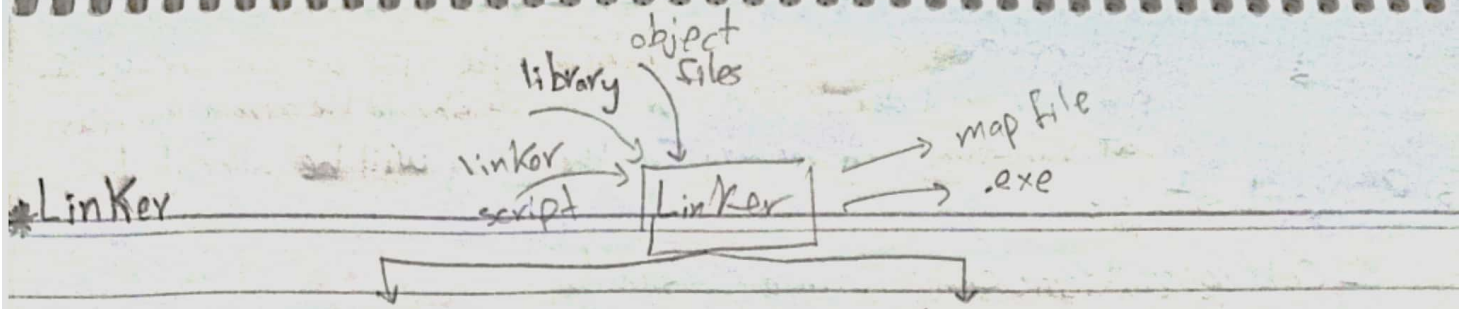
ال linker هو اللي بيحدد → physical

- في مرحلة ال compiler ⇐ debug info من خلالها تقدر أعمل debugger
  - ↳ digging code

- compiler parse only one c file at a time

Symbol table (two sections)

Import	Export
global (extern)	global, static var
Functions (calls) Implemented in other files	functions with implementation
	debug info
(معلومات الفايل عايزها من فايلات تانية)	(معلومات من نفس الفايل ده)



* Linker

Symbol Resolving
check ... symbol ... linker errors

section location (memory)
.o files (relocatable files)
virtual address → physical address
(Linker script) ...

(location counter) . locator ... section location ... linker ...
↳ dot operator (.) → automatically calculates the address
↳ refers to a location in the output section

---

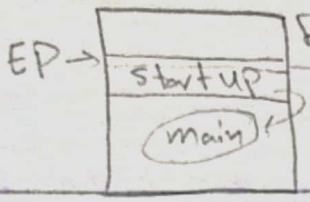
* Compilation Flags :- gcc --help
-E → ... preprocessor (.i)
-S → ... compiler (.s)
-C → compiler + assembler (.o)

---

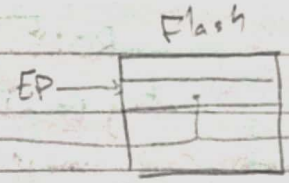
* Booting sequence :-
~~Power/reset~~ → (Entry point) → Instruction life cycle (Fetch, decode, execute
EP

(.bin) → Binary file burn Micro controller
↓
① startup code (.s / .c) → stack ... initialization ...
init stack ...
datasheet ? ... EP Flash
ARM cortex m →
stack Top ... Entry point ... word ...

② Code ⟹ (.text)
③ Global & static (Init to) ⟹ (.data)
④ Global & static (unInit) ⟹ (.bss)



reset section → reset: run startup code



Bootloader (SW for OS)
startup + main func (BL)
Case 2

My Baremetal SW
startup + App.
Case 1

* Case 1

developer ⟹ Entry Point ⟹ Baremetal SW ⟹ startup code + App

PC → entry point → reset section startup

* startup:

1. Initialize stack and allocate space & intialize sp (stack Pointer)
2. copy (.data) Flash → RAM
3. Reserve (.bss) → RAM zero the unintialized data area (.bss)
4. branch / jump (call main)

* Case 2 Boot & load

developer ⟹ Entry Point ⟹ Bootloader (Boot ROM) ⟹ startup code + main func

reset section

start up code ⟹ case 1

* Bootloader:

1. Initializations ⟹ Peripherals (Modules)
2. load ⟹ Flash → OS/SW load → RAM
3. Jump → startup code branch → main

startup reusable memory space Bootloader

Bootloader:

locating → loading → Passing execution

Two Running Modes

ROM mode (case 1)

ROM
- entry point
- executable code + constant data

RAM
- stack + changing data

RAM mode (Case 2)

ROM
- entry point (Bootloader)
- Code Image (OS/SW) ↓
  - ROM ممكن تكون موجودة في
  - SD card - Flash ثابتة

RAM
- executable code
- stack + changing data

* Bootloader ⟹ Code Image —load→ RAM

ROM Mode	RAM mode
- Very simple	- Complex
- require smaller memory	- require bigger memory
- Fixed code address	- Relocatable
- Relative small code	- large code
	- Faster

CamScanner