Date: Page:

lec 21

2 #define : «macro»

- object like macro : 4 syntax

C99:
1. #define identifier replacement list
2. #define identifier(Params) replacement list
3. #define identifier(...) replacement list
4. #define identifier(Params,...) replacement list

Ex: #define x 10 ← يعني أي Identif اسمه x
وبيحط مكانه 10 عن طريق preprocessor
← لي أسرع من int x=10;

→ macro name Rules Same as Identifier Variable.

#define x 10
- int y = x; → int y = 10;
- int x = 50; → int $0 = 50; Error
- int xy = 50; → No → int xy = 50;
- printf("x"); → No
- int X = 80; → No

«Not Recomend» ← Error مفيش #define أخر ال ← متحطش ;

Ex: #define z 10
#define x z
int y = x; → int y = 10;
لو خليت دا قبل دا عادي مفيش مشاكل.

ROX

Date: Page:

Notes

1. `#define x 10`
   `#define x 20` → "warning" → redefinition
   ↳ قيمة x بعد السطر دا هتتبدل لـ 20

2. `#define x 10`
   `int y= x+10;` → y = 20 = 10+10
   `#define x 20` → "warning" → to avoid use #undef
   `int z= x;` → z = 20
   ↳ بيعمل remove لـ definition
   `#define x 10`
   `#undef x`
   ↳ عدد "i" ملفات = = عدد ملفات "c"

hint → `gcc -E main.c -o main.i` → علشان تقف عند file "i"
`gcc -S main.c -o main.s` → علشان تقف عند Assembly file

enum	V.S	#define
→ text replacement by Compiler at Compile stage		→ text replacement by preprocessor at preprocessor stage
→ must be Const integers if use float → Syntax Error		→ we Can use float, Sentance any thing
→ Space = integer Size Compiler dependent		→ No memory Space

Date:

Page:

enum v.s. #define

→ you Can use inside Switch — → you Can't use inside Switch

→ Same value for different names — → سطر واحد قيمة واحده اسم واحد

→ $[-2^{n-1} : 2^{n-1}-1]$ — → any Value

---

typedef v.s #define

→ By Compiler — → by preprocessor

→ limited to datatypes — → used with types and Values & So on

unsigned Char ⇄ uint8 ⮐ Symbolic name

→ typedef struct student * ptr; ⮐ Semi Colon لازم — → #define ptr struct student*

→ ptr* m, n; ⮐ two pointers — → ptr x, y;

ليه لأن دا بيعرف datatype جديده ←

struct student *x, y;

x pointer ⮐

"not ptr" Struct student y ⮐

→ Function Like Macro

⮑ parameterized macro

Ex: #define Add (x, y) x + y

Date: Page:

int z = Add(x,y) ⟶ int z= x+y
No space

hint → #define Add (x,y) x+y
space
int z= Add (3,4) ⟶ int z= (x,y) x+y (3,4);

→ Concatination Operator '\' Back slash
يستخدم علشان نعمل Concate للسطور كأنها سطر واحد

#define x \
5
= #define x 5

\ ⟶ used with function like macro ⟶ Readability علشان تزود

Ex: #define Sun(—,—) —— \
—— \
——
——

بدل ما يبقى سطر واحد طويل وشكل الكود يبقى وحش

Advantages of Function Like macro:
1- Fast in execution ⟶ "No Context switch"

Dis Advantages:
1- Increase Code size (context) ⟶ text replacement لأنو بيعمل
2- No type checking ⟶ مثلاً لو فانكشن مبترجعش حاجة وحاولت ترجع float الـ Func like macro مش هتتبهك وهيبوظ memory
3- difficult to debug ⟶ text replacement لأن حصل
——

Date: Page:

when to use Function like macro:

- لو مش هتنفذها كتير
- لو Time مهم عندي

In Embedded:

- "Bit_Math.h" → ملف بتعمل عمليات Funcs على مستوى Bits
  Set_Bit, CLR_Bit, Read_Bit, Toggle_Bit

→ macro will make your Code professional but hard to debug.

(...) → Any thing else. → user معناه بناخد قيمة معينة من
__VA_ARGS__ ← Var بتخزنها في دا اسمه

Ex: #define func(...) printf(__VA_ARGS__)

```
int main() {
    func("hello\n");
}
```

← اي حاجة هتتحط مكان ... هيشيلها ويحطها في __VA_ARGS__

Syntax 4: Ex: #define func(x,...) printf(__VA_ARGS__,x)

func(x,"x=%d"); ≡ printf("x=%d",x);

⇒ Predefined Macros "Built-in":

Ex: __DATE__ ,..... ممكن تستخدمها في الكود بتاعك

- __FILE__
- __FUNC__
- __LINE__

→ تقدر من خلالهم تجيب معلومات الـ Code

Date: Page:

3 Conditional Directives:

1. #if Condition 1
   #elif Condition 2
   #else
   #endif

PreProcessor → #if

← زيها زى if العادية مع بعض الفروق زى ان دى بتتنفذ ع Preprocessor

← علشان اختم بيها

← اللى بيخش مرحلة Compile السطور اللى تحققت فقط (الشرط اللى اتحقق بس) الباقى اعتبره Comment

if	V.S	#if
→ Compile		→ preprocessor
→ check vars		→ Check macros not vars
		Ex: #define x 10
		#if x > 70

usage:

1. make Comment → #if 0 ... #endif } Comment multi_line
2. Configration → driver بتاع

Date: Page:

2- #ifdef , #ifndef

if define] └→ if not define macro] لو macro]

For Ex: #define x └→ used in header file guard

#ifdef x

#else

#endif } → Comment

header file عشان يحمي ← include من انك تعمل أكتر من مره

#How : #ifndef file_macro ~ _FILE_H_ Naming Rule]

#define file_macro For Ex: UART_H_

#endif

3- #if defined , #elif defined

logical operator ||, &&, ! ←

Ex: #if (defined x) && (defind y)

#endif

4 #error , #warning

Stop Compilation] └→ just warning

with Error msg] └→ used with #if not if

└→ check @ run time

Date: Page:

Ex: #if (_____)
#error "_____"
«√»

if (_____)
#error "_____"
↳ هيتنفذ حتى لو if ← false

---

5 Stringification && Concatination:

1- stringification «#»:
بيحولها لـ string
Ex: #define printf(x) printf(#x)
↳ printf(muhammad); ↳ strinize operator
↳ printf("muhammad");

2- Concatination «##»:
بتعمل Concatinate بين حاجتين
Ex: #define Conc(x,y) x##y
↳ taken-pasting operator

int x= Conc(3,8);
↳ x = 38

---

use () with macro #define sum(x,y) (x+y)
↳ علشان تجنب المشاكل لأن دا text replacement

Ex: #define sum(x,y) x+y → (x+y) → best

printf("%d", sum(3,4)*2);

Expected: 14 ,Actually: 11
@preprocessor: printf("%d", 3+4*2); → 11
↳ That's why we use ()

Date: Page:

[*] usage of preprocessor directives:

1- Configurability ↔ حاجة تعدلها

Ex: #define SIZE 256 بمجرد معدل ال Size
Char arr[SIZE]; بقيت السطور تتعدل لوحدها
اللي مستخدم فيها Size

2- Readability → more professional Code

3- portability

Ex: typedef unsigned char uint8; you Can use #define with problems

→ #line → #line "20" → بتخلي السطر اللي بعدها رقم 20
int x; → سطر رقم 20

#pragma : Compiler directive ← حاجة بعتها لـ Compiler علشان يعمل حاجة معينة

Usage:

1- #pragma optimize ("", off)
↳ "No optimization"

2- #pragma once → replace file guard

hint → #pragma → Compiler dependent

__attribute__ (( )) ← وبعض Compiler بتبقى

← طبعًا يفضل تستخدم file guard علشان متبقاش تحت رحمة Compiler

Date: Page:

3. #pragma startup [Priority]
#pragma exit [Priority]

→ default priority 100

Priority optional
↓ int [64:255]

Startup
64 is the highest priority

Exit
64 is the least priority

→ C libraries priority is [0:63]

Ex: #pragma startup func 105
Exit

Ex: Void func1();
Void func2();
#pragma startup func1 105
pragma startup func2 → 100 default
pragma exit func2 → 100 default
pragma exit func1 105

105 > 100 «startup»
105 > 100 «exit»

```
int main(){
    printf("main"); }
Void func1(){
    printf("func1");}
Void func2(){
    printf("func2");}
```

O/P
func2
func1 } startup
main
func1 } exit
func2

priority حسب على

Date:

Page:

4- Add memory - Section:

#pragma region name = "____" origion = 0x.... (Address)
size = 0x....

Var ⟶ store @ not default memory Segment يعني أي غير

lec 22

Embedded C

#Contents:

→ Tool chain
→ Make file, Linker script, startup Code
→ Booting Sequence "Embedded linux"
→ Boot loader
→ memory layout
→ gdp [Commands, Cract]

[*] Tool chain:

→ Definition:

مجموعة من ال Blocks ⟵ .C File ⟶ [tool chain] ⟶ executable file

بيعدي عليها علشان ع الاخر Code أحصل على ex file

tool chain ↳ Compiler + Assembler + Linker + debugger + Libraries + preprocessor + #: helper programmer

↳ Binary utilities

من خلالها بتعمل analysis لـ Binary

Date: Page:

⇒ Types of tool chain

Cross tool chain

I/P → T.C → O/P

بعمل Compile لـ Code عالجهاز target والـ O/P بيرن على Arm, AVR, ...

Native tool chain

I/P → T.C → O/P

O/P بنعمله run على نفس الجهاز اللى عملت Compile لـ Code عليه نفس Arch

---

Tool Chain
- C Code
- free
- Commercial «mongey»

Systems:

- Build System
- Host System
- Target System

[*] Build System:

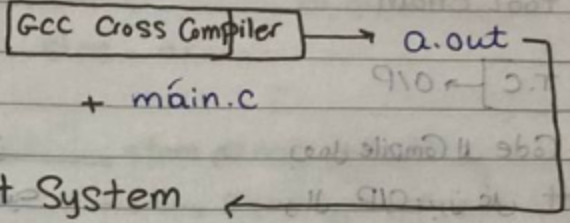
→ Source Code of tool chain.

Tool chain → GCC Native → GCC Gross «Compiler»

← الخطوة دى مش بنعملها لأن الشركات ممكن تقفل open source ← Source Code فلما تعمل لـ Compile ← Source Code بتاعتك تطلع Tool chain بتاعت GCC Native أو الشركات تنزل Tool Chain جاهزة زى Arm Cross T.C

Date: Page:

[*] Host System:



target system دا اللى بينزل على

CMD → "GCC":

- → gcc main.c —out→ a.exe
- → gcc main.c -o main.exe
- → gcc main.c -I /usr/share. ← header file لـ include بعمل
- → gcc -wall main.c → Enable warnings. ← بتوقف لما يحصل warn
- → gcc -Werror main.c → Enable Errors. ← لما يحصل Error
- → passing options using file
  - ↳ gcc main.c @optionsfile ← بحط الاوامر فـ file واديهالو
- → gcc --help

Libraries

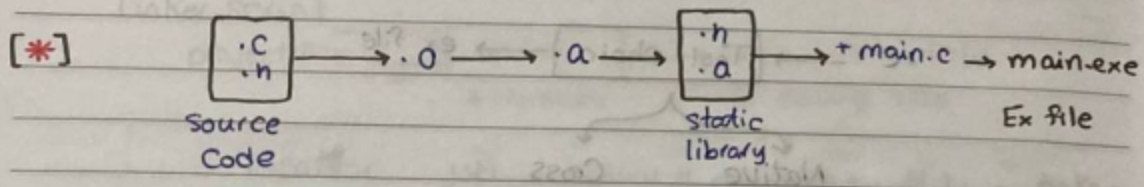
- static ← مش بتشوف «.c» ← يعني مش بيشوف source code
- dynamic ← بتشوف «.c» عادي

← يعني مش هديله «.c» هديله «.o» أو «.a» ← object file و «.h»

دا اللى هيخش فـ linker علشان كدا لازم تعمل كل حاجة extern لـ user

Date: Page:

→ How to make static lib?



→ gcc -c file.c -o file.o ← علشان أوقف ال Compiler عند object file

to get static lib:

ar rcs file.a file.o

to get executable file gcc main.c file.a -o main.exe

#CMD:

→ ar r file.a o_file.o — يضيف object file في library

→ ar d file.a o_file.o — يحذف object file من library

→ ar t file.a — علشان تشوف object files اللي في library

→ ar x file.a — تعمل extract لـ lib وتخرج كل object files

---

hint → Arm Cross Tool chain

→ GNU GCC «Commonly used» → free, open Source

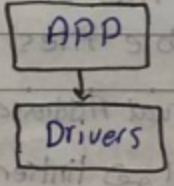
→ arm cc «Commercial»

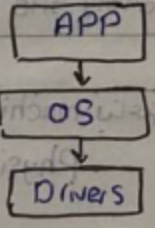
Arm Tool Chain

↓

arm-none-eabi-gcc ✓✓ — arm-linux-gnuabi

→ Baremetal APP — → OS APP



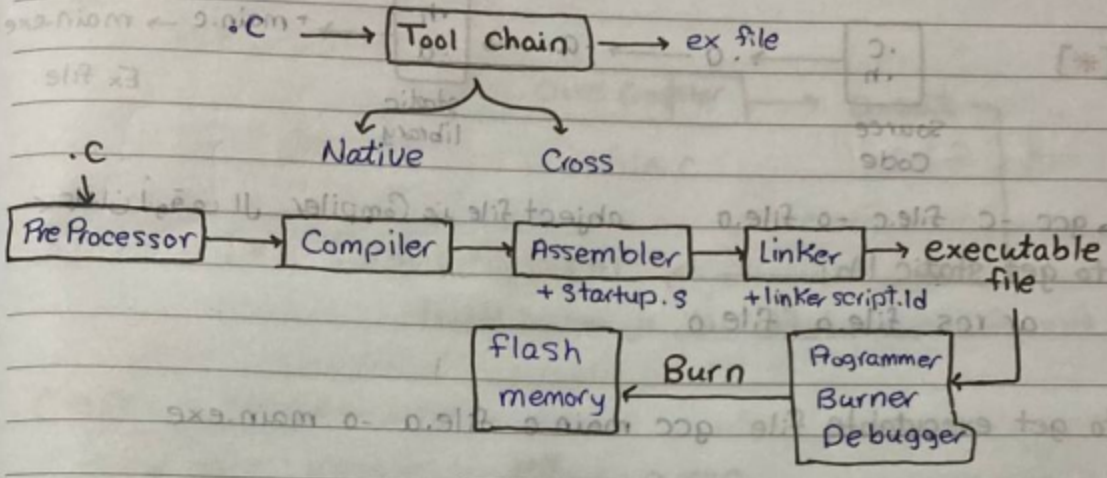


Date:

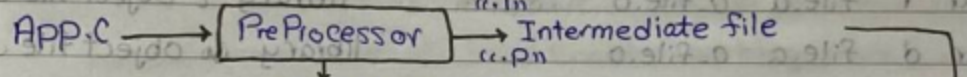
Page:

Lec 23

"Compilation process"



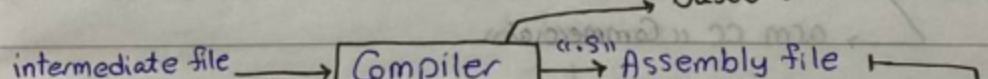
1- preprocessor stage:



عددها متساوي للي داخل «.i» == «.c»

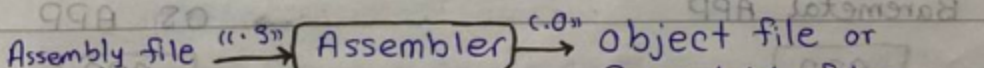
text replacement ⟶ preprocessor directives

2. Code Generation Stage "Compiler" : "Based on Arch"



لكل Arch : Assembly Inst ظاهرة بيه ممكن تتشابه أو تختلف مع Arch تاني

3. Assembler Stage:

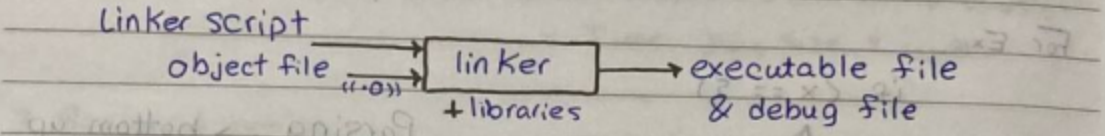


يعني ليه متعملوش mapping على machine بتاعتي : Virtual Addresses.

هو اللي بيحول Virtual Addresses لـ Physical Addresses.

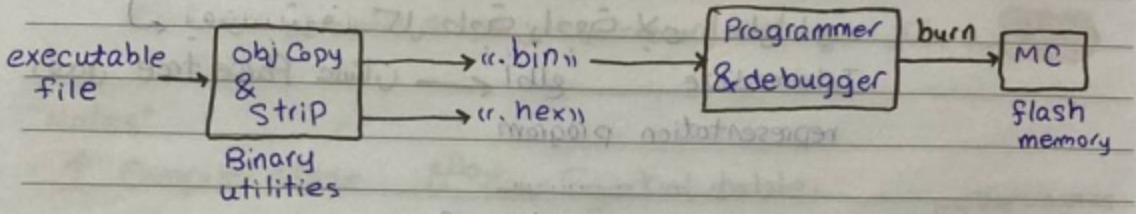
Date: Page:

4_linker stage :



locator ← دا اللى بيعرفة تقسيم الـ memory من خلال linker script

Address map ← locator



[*] Code generation stage "Compiler" :

- Front end stage
- Middle stage
- Backend stage

1_frontend stage

- Source code parsing
- check Syntax Error

made by tokenization : ← يعنى Compiler بيحاول يعرف الـ Code

↳ generate tokens → Keywords & operators & Identifiers

Syntax Analysis ← دا

Syntax Analysis : بيشوف الـ Code بتاعك متوافق مع Standard بتاع C و X ولا

Error ← لو مش متوافق

lexical Analysis ← بيحصل tokenization ودا اللى بيطلع tokens ودا اسمه

ومن خلال إنك تشوف التوافق مع standards دا Syntax Analysis .

O/P: Parse tree → IR Programe

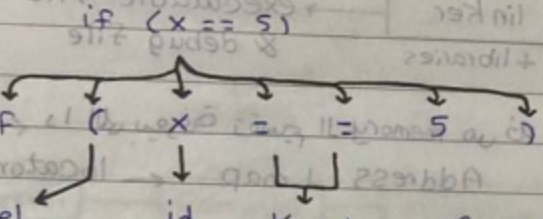
Symetic table

Date:

Page:

Parse tree ← مرحلة بتيجي بعد tokenization

For Ex



Parsing → bottom up
→ top down

Parse tree ~ دا اسمه

يعني بشوف كل حاجة راجعة لإي أطلعلها اي

يعمل Parse tree علشان ← اطلع Intermediate representation program

---

2. middle stage: → Symantic Analysis
→ Optimization

- Symantic Analysis: logical structure «Data types»

بيحاول يفهم ال Code وبعدها يضيف بسطور على «IR»

hint → warning ~ for ex pointer to char → point to int without Casting ← دا يعمل Symantic Analysis ع.

- Optimization:
  - → decrease Code size and execution time and memory size.
  - → Multi level process
    1. Dead Code "unreachable Code" → remove
    2. inline expansion of function func → inline func
       Optimizer شغل ex time فيقلل Context switch يقلل
    3. Register Allocation «GPR» → Bank registers
       ملهاش تعامل من خلال Address لانها جوة Processor.

4- loop unrolling → يحول ال for loop مثلاً لعدة سطور تحت بعض
هيزود ع size ويقلل Time

3- Backend stage :
- → Code generation : Convert into Assembly.
- → Memory Allocation : .data , .bss , ... بيعرف memory section

hint → ممكن تعمل memory sections من خلال ال linker script او #pragma

---

"Notes"

1- Compiler stage بيطلع → Symbol table.
└→ Symbols
- → Var-names
- → func-names

Symbol table

Symbol	Address
	•

→ Virtual Address

→ دا لكل File
علشان linker
لما يحتاج حاجة Address

2- Compiler stage → debug info
└→ to use debugger → digging

Code ← debugger مش بيفهم 0,1 فيحتاج حاجة تفهمه ال Code

hint → optimizer : ساعات ع Vars بيحذفها من ال Code فال debugger محتاج حاجة تشاورة زي debug info

---

3- Compiler → parse only one c File at time.

Date:

Page:

Symbol table

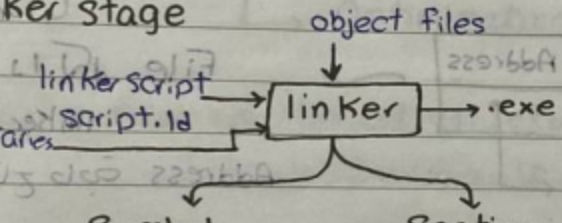
Symbols

- Import ← المعلومات اللى File دا عاوزها من Files التانية
  1. global & static vars «extern»
  2. Function «Calling» Code
  - Compiler طبعًا لو ملقاش وده مرحلة linker هيديك linker Error ووصل لـ
- export ← المعلومات اللى File دا عاوز يوصلها لبقية الـ Files
  1. global & static vars
  2. Functions name
  3. debug info
  - بتوع File نفسه

Modern Compiler : Compiler و Assembler بيبقوا Block واحد ← hint

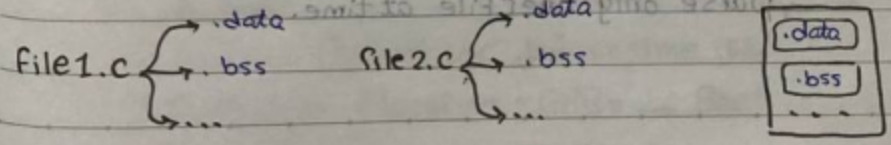
---

[*] Linker stage



- Symbol Resolving ← هيشوف لو File محتاج Symbols موجودة في Files تانية وبعضها بـ link بين Files
- Section location ← دي تقسيمة الـ Memory بناءً على linker script ← physical Address

linker : هو اللى بيحول Physical Address لـ Virtual Address بمساعدة الـ linker script اللى بيعرف memory layout



يجمعوهم كلهم في file واحد

ROXI

Date:

Page:

hint ← ممكن تغير أسماء الـ Sections بس يفضل تسيبها زي ما هي الـ standard

linker ← locate Section باستخدام locator ← location Counter

└→ dot operator (.) → بنحدد أسماء الـ Sections والـ Boundaries بتاعتهم

Is linker Important if you have one file «main.c»?

- yes linker → section location
  └→ physical Address

---

hint → Compilation flages: gcc --help

- -E → preprocessor only (.i)
- -S → Compiler only (.s)
- -C → Compiler + Assembly (.o)

← ممكن نعمل file.map ← linker بيطلعه بيبقى فيه Symbols بـ Physical Address.

---

lec 24 "Booting Sequence"

when → power on / reset of MC → Entry point

← عبارة عن Address اللي بيبدأ من عنده في flash

تعرفه من "data sheet"

Instruction-life Cycle: «fetch - decode - execute»

← بيبدأ ينفذ الـ Code من عند ها

(.bin) → binary file —burn→ MC

└→ startup Code → Code بيتنفذ قبل main

└→ Assembly or C ← في الغالب بيبقى Assembly

← لأني لسه معملتش Init لـ Stack يعني مقدرش أعمل Functions

← أقدر أكتب startup بـ C ← لو فيه حاجة عملتش Init لـ Stack

Date:

Page:

ex → Arm Cortex ⇒ Init → stack → من خلال اول بيانات اول حاجة

stack بتاع top بيحطها في [redacted]

EP ← من اول

stack ل Init قبل stack pointer بيحطها في

(.bin) → Code متخزن في → (.text)

→ global & static (Init ≠ 0) → (.data)

→ global & static (Init = 0) → (.bss)

flash

EP → startup

main

hint → Assembly Func :

label :

Code → Func in Assembly

اول حاجة reset section

reset : → resst section

→ run اول حاجة بيحطها

→ reset بس لازم يبقى اسمه متغير

standard ده

flash

EP →

Boot loader

OS بدون sw

startup + main → for Boot loader

your Baremetal sw

startup + App

Date: Page:

1- devoloper → Entry point → Baremetal SW
startup + APP

Program Counter → دا اللي بيشاور على next Inst اللي عليها
Startup بتاع reset section ← EP الدور في التنفيذ فبيشاور على

تحتوي على الحاجات اللي startup بيعملها
1- Init → Stack
2- Copy (.data) from flash to RAM
3- (.bss) → مش بيبقى ليها مكان في flash لأنها معروفة
Reserve RAM ↗ بـ Zero فبيحطها في RAM على طول
بيعمل حاجات MC ↙ يعني أثناء ال load time ملهاش مكان في flash
وأثناء ال Run time مكانها في RAM
4- branch to main → Main

2- developer → Entry point → Boot loader ex: Boot ROM
Startup + main

startup Code : reset section
هيعمل نفس الحاجات بس لما يجي (Boot loader) main عند
load: from Source to distination ينقل

1- Boot loader ← Init : Peripherals «modules» ← Static دا اللي بيقوم مع Kernel
الحاجة اللي محتاجها يعني ← For Ex: Init SPI ← dynamic
ex : load 2 ← flash ← RAM
OS / fw
Baremetal SW ← Embedded linux
3- Jump ← Startup بتاع SW اللي هيعمل Branch
main (SW)

و هنا يظهر Memory فاضية بتاعت Boot loader لأنو خلص.

Date:

Page:

Two Cases: two Running modes

ROM mode → Baremetal SW

RAM mode → Boot loader

Baremetal SW

ROM
- → Entry point
- → Executable Code
- → Const data
- ↳ علشان كدا RAM mode اتسمى

RAM
- → Stack
- → changing data

Boot loader

ROM
- → Entry point
- → Code Image
- SW / OS
- ↓ مش شرط يكون ROM ممكن ← SD card, Ethrnet, flash

RAM
- → Executable Code
- → stack
- → changing data
- ↳ علشان ex Code في RAM اتسمى RAM Mode

ROM Mode	V.S.	RAM mode
→ very Simple		→ Complex Code
→ Small memory		→ large Code
→ fixed Code Address		→ Relocatable mode
→ Relative to small Code		↳ ممكن توصل SD مثلاً 64 GB
		→ fast → لأنو شغال من RAM
		و RAM أسرع من ROM

---

hint
- → Embedded linux → RAM mode (Bootloader)
  - ↳ لأن حجم ال Code بيبقى كبير.
- ↳ Boot loader → has its own startup Code.
