.c → .i

preprocessor directives

1. # include
   to include header file ".h"

include < File.h >
Standard
built in libraries

include "file.h"
or path
then current directory
standard

"path"

Absolute
pC

Relative
current directory

Notes 2-
* step forward #include "../../"
* step backward #include "../../"

2. # define "macro"

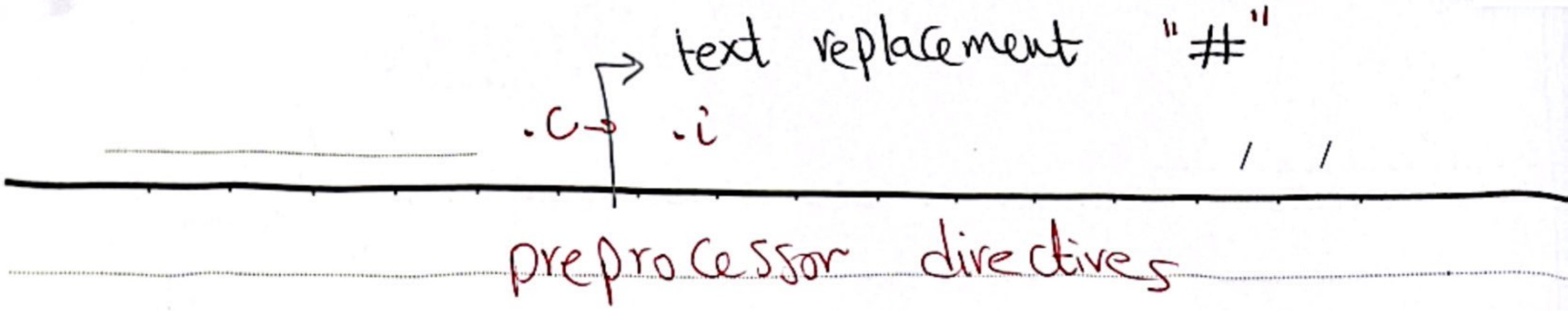
object like macro

define Name value
"replace any name by that value in preprocessor stage"

function like macro
"parameterized macro"

define name (parameters) replace

ex2- # define Add(x,y)
x+y no



① # include

to include header file ".h"

include < File.h >
Standard
built in libraries

include "file.h"
or path
then current directory
standard

"path"

Absolute
PC

Relative
current directory

Notes 2-

* step forward #include "../"
* step backward #include "../../"

② # define "macro"

object like macro

define Name value
"replace any name by that value
in preprocessor stage"

function like macro
"parameterized
macro"

define name
(parameters) replace

ex2- # define Add(x,y)

Notes :-

* #define x 10
  #define x 20 > warning - redefintion

to avoid warning :-
#define x 10

#undef x

#define x 20

* #define ptr struct student *

ptr x,y ; replaced by

struct student * x , y ;
pointer variable

* if
#define Add [(x,y) x+y]
space
replace Add with the whole bracket

* Concatination operator '\'
back slash

#define x / = #define x 5
5

→ used with function like macro
#define _(_,_) \
\

adv
fas

disadv

(...) anything else

takes → value stored → var (__VA_ARGS__)

ex # define karem(...) printf(__VA_ARGS__)
void main()
{
karem("helloo");
}

Syntax 4:-

#define func(parameter,..) replacement

pre defined macros:-

(3 Conditional

(1 #if condition1 — if first condition is true else will not enter compilation

#elif Con 2

#else — remained code will not enter "Comment code."

#endif

Note 2-

* #if check macros not variables → run time

ex 2-

```
#define x 10
#if x > 70
```

Usages:-

(1 Comment — #if 0 ... #end if

(2 Configration — driver

(2 #if def — if macro has been defined

#if ndef — if macro has not been defined

usage 2- header file gaurd → "to avoid multiple definition"

(3) #if defined & #elif defined

ex

#if (defined x) && (defined y)

Can use logical operations

(4) #error → stop compilation
error messege

#warning

used with #if

note:-

if (__)
#error "__" → will be execute even the condi is false

(5) stringification & Concatination
##

(1) (String / strinize) will put it in douple qoates

(2) Conca (taken pasting operator)

ex:- #define Con(x,y) x##y
(xy)

Usages 2 -

1) Configurability ex #define SIZE 256
char arr [SIZE];

2) Readability

3) portability # define u8 unsigned char

hints 2 -

#line determine no. of line after it

#pragma compiler directive

Tool Chain :- tool chain

.C → [preprocessor + Compiler + Assembler + linker + debugger + libraries + no-helper programmer] → File

(Binary utilities)

ex "objcopy, objdump, readelf"

Types :-

- Cross: Compile on PC then run on "target"
- Native: Compile and run on same device

Systems :-

- Build system
- Host system
- Target system

Source code of T.C → GCC Native

GCC Cross "compiler" + main.C → a.out

"app"

بناتها جاهزة

CMD gcc

1. gcc main.c a.exe
2. gcc main.c -o main.exe "to Name"
3. gcc main.c -I /user/ -- path
4. gcc -wall main.c enable warnings errors]sto
   ~~-wall e~~ -werr
5. pass options using file
   ↳ gcc main.c @optims file

gcc --help

Libraries :-

Static → with no source code
object files ".a" ".o"

dynamic

h → Fun name → project → T.C

→ linter →
→

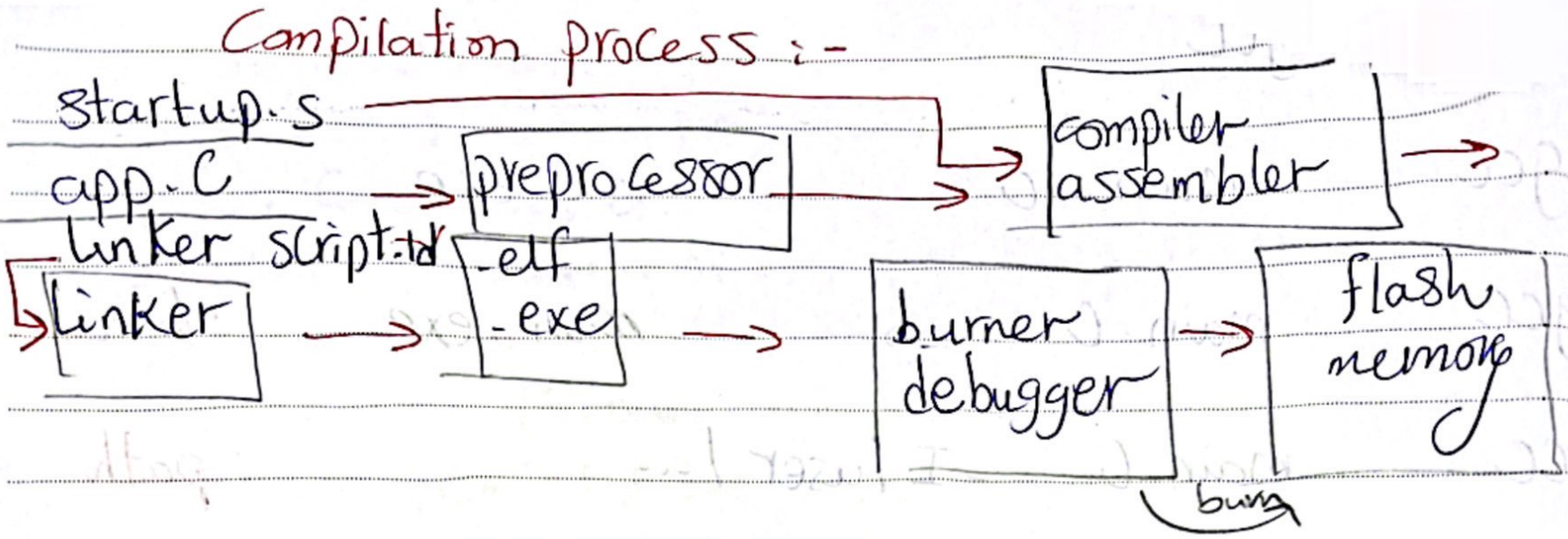
hints :-

ARM Cross tool chain

GNU GCC

arm CC

Compilation process :-



app.C → pre processor → .i / .p → compiler

"text replacement" "code generation stage"

→ .s → assembler → .o →

"based on device Arch" assembly files

objeet files
Rebocatable Files
has virtual address
"0, 1"

Lib → linker ← Linker script

linker → physical address → .exe debug files →

devide memory

Bin utalaties → .bin / .hex → debugger →

"obj copy"

flash in MC

Compiler "Code generation stage" :-

Front end

1. Source code parsing
2. Check syntax errors

how?

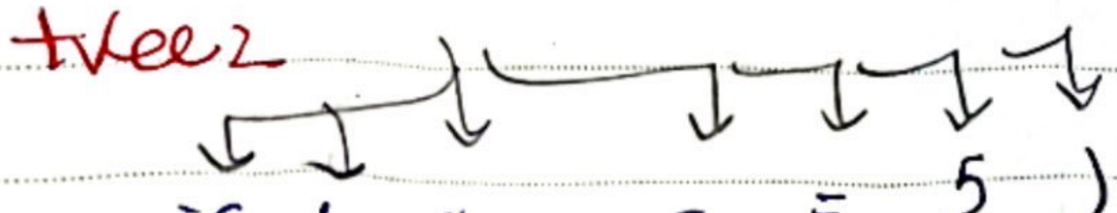
tokenization → tokens → keywords, operators, Identifers

Syntax analysis
check with standard
↓ if not identical
error

output: Parse tree
bottom up / top down

ex: if (x==5)

tree:



→ IR

Middle

1. Symantic analysis
2. optimization

1 → check logical structure
+ IR

hint: warning if pointing to different datatype

2. Why?
Code size ↓
exe time ↓
memory size ↓

How?

1) dead code → remove
ex: after return

2) inline expansion of function to reduce time

3) Register Alloc
"GPR"

4) loop unrolling

Back end

1. Code generation convert into assembly
2. memory allocation sections
ex: (.data .bss)

compiler instruct Linker using symbol table

Symbols	Address
variables	
function	

→ virt

→ static
→ global

Notes :-

2. Compiler → debug info → debugger
digging code
help debugge. understand os and ls

3. compliers parse only one c file at atime
then linker link them together

Linker :-

- symbol resolving
  Find symbols in diff files
  has to be in import section of a File and export of another one
  if not → error
- section location
  chang virtual address in object files to physical ones
  using linker script
  locator (location counter "dot operator" (.))

Booting Sequence :-

Power Reset → Entry point (in) → Flash

data sheet → address in

start up Code :-

- .s → assembly
- .c → C code

Intialized stack by "ARM cortex"

.bin (binary file) — burn → MC

(1) startup code

(2) Code → (stored) .text — have same size

(3) global - static → .data
Intialized ≠ 0

(4) global - static → .bss
Intialized = 0

* start up code must be in Entry point before ma
code

hint :-
Assembly func

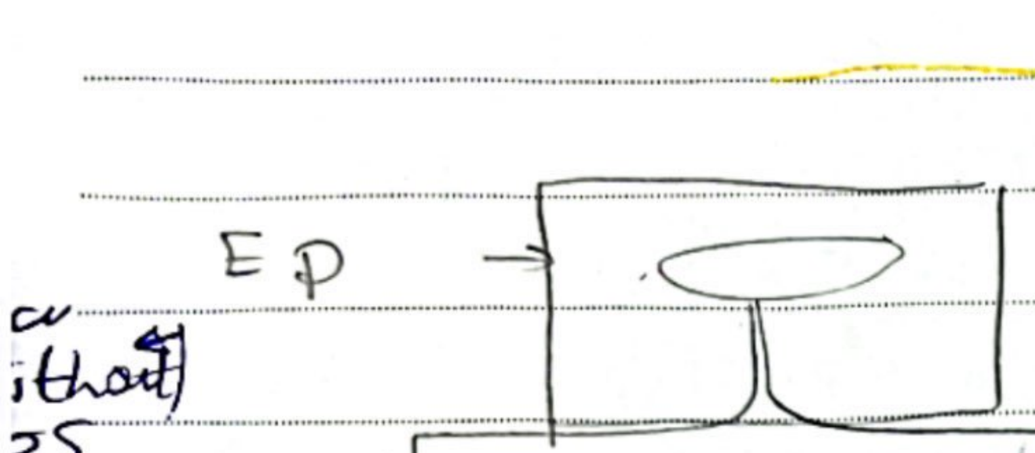
label :

reset section

to write in assembly

reset =

first to run in startup code



Boot loader (startup_main)

your Baremetal SW (startup_App)

Entry point Boot loader PC → Entry point → reset

ex: Boot ROM

what startup do?

→ reset

→ Same 4 functions

then

enter main of boot bader

(1) Intialize stack

(2) Copy (.data) from Flash to RAM

(3) .bss has no pkce in Flash ! reserve in RAM

(4) brach / jump → main

→ load: (1) Initialize modules

ex: SPI

(2) load flash → OS / SW

load RAM

(3) Jump from sw startup → main

→ SW

Running modes :-

ROM mode
Baremetal

- ROM
- RAM

Entry point
exe code
constant data

stack
Change data

- Simple
- require smaller memory
- fixed address
- Relative small code

RAM mode
boot loader

- RAM
- ROM

exe code
stack
Change data

Entry poi
Code ima

- Complex
- large code
- Relocatable mode
- fast