EMBEDED C

Preprocessor directive:

1-#include

2-#define

3-conditional directive

4-#error /#warning

5-stringification and concatenation

- ❖ Preprocessor >>> text replacement >>> # (only preprocessor understand it)
- Except: #pragma >>> compiler directive.

1-#include:

- To include header file (.h)
- How to include it ???

a-#include <file.h>

standard library (built in) (هيروح يدور فيهم)

b-#include “file.h” or #include “path”

current directory if he does not find the file then he searches in standard lib.

#include “path”

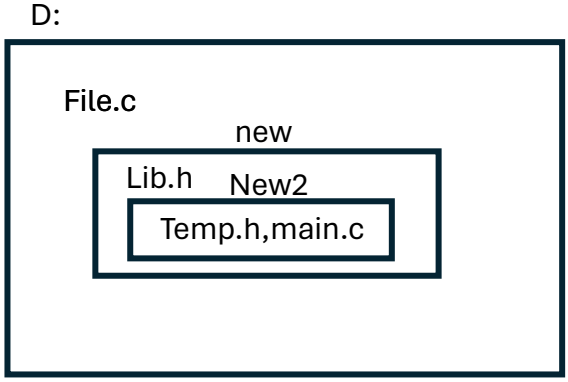
Path has 2 types :

a-absolute path: pc (don’t use it)

b-relative path: related to your working directory.

EX: file.c >>>>>> temp.h #include”new/new2/temp.h”

Main.c >>>>>lib.h #include”..lib.h”



2-#define(macro):

has 4 syntax

1-#define identifier replacement list

2-#define ident(param) replacement list

3-#define ident(...) replacement list

- (...) >>> anything else
- #define karem(...) printf(__VA_ARG__)

4-#define ident(param,...) replacement list

- #define fun(a,...) printf((__VA_ARG__)

Types:

a-object like macro.

- #define name value ex: #define x 10

b-function like macro(parameterized macro)

- #define add(x,y) x+y int z=add(x,y); >>>>> int z=x+y

Hint : #define add(space)(x,y) x+y

Int z=add(3,4)>>>>>> int z=(x,y)x+y (3,4)

Advantage :

1-fast.

disadvantage :

1-code size.

2-no type checking.

3-difficult to debug.

- When to use it ??? if it will not repeated
- In embedded: “BIT_MATH”

Macro name rule:

1-start with letter or underscore

2-without spaces or without special characters(except _)

- It is recommended that name is uppercase.
- Not recommended to put ';' at the end .

What will happen if ?????

#define x 10

1-int y=x >>>> expansion >>>> int y=10

2-int x=50>>>> expansion>>>>10=50 (compilation error)

3-int xy >>>>>no expansion

#define z 10

#define x 10

1-int y=x >>>>int y=10

Notes:

define x 10

#define x 20

- Will give warning (any line of code after (# define x 10

)will operate that x=10 and any line of code after #define x 20

)will operate that x=20 and).

>>To avoid warning

define x 10

#undef x

#define x 20

- (.c)files=(.i)files
- concatenation operator '\' backslash used with function like micro

#define x \

5

===#define x 5

Macro :

1-Professional code

2-Difficult in debugging

❖ Different between #define and Enum

#define	enum
• Text replacement in preprocessor stage • We can use float sentence. • Not Used inside switch • No memory space. • Only one value and one name • Any value	• Text replacement in compiler stage • Must be const int (not float) • Use inside switch • Int size • Sane value to different names • -2^n : 2^(n-1)-1

❖ Different between #define and TYPEDEF

typedef	#define(macro)
• Text replacement in compiler stage • Char >>>> u8 used for types only • #define struct stud *ptr; ptr * m,n;(two pointer)	• Text replacement in preprocessor stage • Used with values as well • #define ptr struct stud* Ptr x,y;(will replace x by pointer to struct and y is a variable of type struct stud)

3-conditional directive:

a-#if cond1

#elif cond2

#else

#endif

- We can use it to comment code

if	#if
• compiler • Check variables.	• preprocessor • Check macros not variables.

Usage:

1-comment

#if 0

#endif

2-configuration

b-#ifdef , #ifndef

- use in header file guard.

c-#if defined ,#elif defined

4-#error(stop compilation,error message) **, #warning** (only warning)

- Used with #if not if as if check in runtime

5-stringification (#) and concatenation(##)

- stringification (#):convert it to string

EX: #define printf(x) printf(#x)

Printf(.....) >>>>> #==” ”

Concatenation:

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

Int z=conc(3,8) >>>> int z=38

Hint:

```c
#define max(a,b) a>b?a:b

int main()
{
int x=9,y=7,z;
z=max(x,y)*2;
printf("%d",z);
}
```

Output :9

To avoid that use brackets

```c
#define max(a,b) ((a)>(b)?(a):(b))

int main()
{
int x=9,y=7,z;
z=max(x,y)*2;
printf("%d",z);
}
```

Usage of preprocessor directive:

1-configuration

2-readability

3-portability

Hint : #line 20 will consider the next line to be 20

#pragma

Usage:

1-#pragma optimize (“”,off) >>>>> no optimization

2-#pragma once >>>>>> replace file guard

Note: compiler dependent but file guard is independent

3-#pragma startup[priority], #pragma exit[priority]

4-add memory section

- #pragma region name=” ”

Tool chain:

Define: .c → tool chain → exe file

Consist of:

Preprocessor+compiler +assembler+linker+debugger+library +helper programmer (binary utilization) ex:objcopy,obj dumb,readelf

Types:

a-cross: compile at pc and the output run on the target hardware.

b-native: compile and run on the same machine .

- tool chain >>>> c.code

systems:

source code of tool chain >>>>>gcc native >>>gcc cross(compiler)+main.c>>a.out

build system host system targetsystm

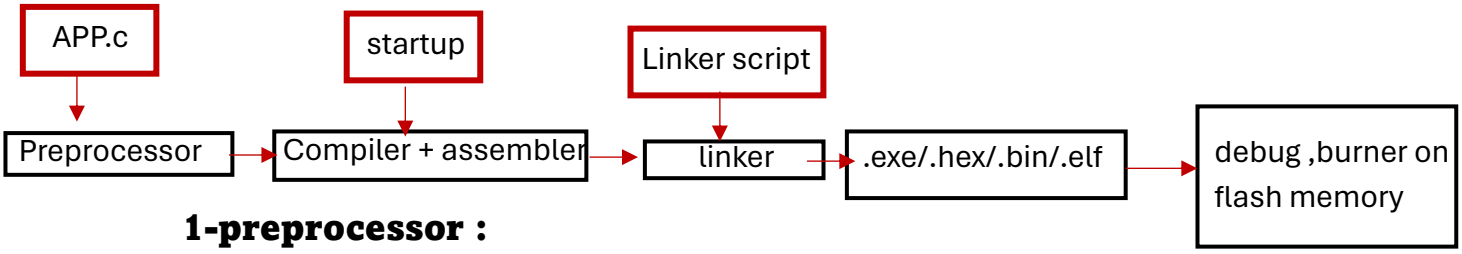
libraries:

1-static: we cannot see .c (object file(.a))

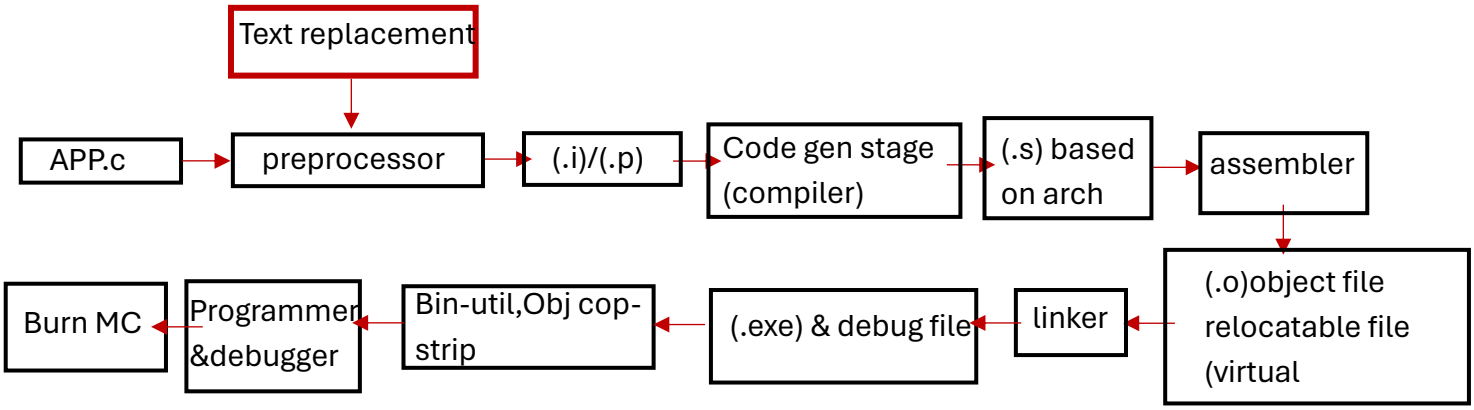
2-dynamic: we can see .c

how to ??? .c/.h >>>>> .o >>>> .a (static library)

Compilation process:



1-preprocessor :



2-comiler:

a-frontend stage: source code parsing + check syntax error

- syntax analysis /tokenization >>>> tokens
- o/p>>>>> parse tree >>>> IR program

b-middle stage:

- Semantic analysis: check logical structure.
- optimization: why???

Reduce code size /exe-time/memory size.

How?? Has multilevel process

1-Remove Dead code.

2-Inline expansion of fun

3-Register allocation

4-Loop unrolling

c-backend stage:

- code generation: convert into assembly from code language.
- memory allocation

note:

- compiler >>> symbol table (variable name/fun name)
- compiler >>> debug-info >>> debugger >>> digging code
- compiler >>> parse only one c file at a time

symbol has two sections:

a-import: global & static variable which need from another file/fun (call)

b-export: global & static variable/fun (implement)/debug info

3-linker :

a-Symbol resolving

b-section location: convert virtual address to physical address by using linker script

it does section location by the help of locator(location counter) .operator

hint :

compilation flags

-E >>>> preprocessor stage (.i)

-s >>>>compilation (.s)

-c>>>> compile +ass (.o)

Booting sequence:

Power on /reset >>>> entry point

Inst-life circle >>>(F,D,E)

Binary file contain :

a-startup code (.s/.c)

b-code (.text)

c-global & static >>> .data (init !=0)

d-global & static >>> .bss (uninit)

boot loader: software without os(startup code +main)

two cases :

case 1: baremetal SW

developer >>> EP >>> baremetal sw (startup+APP)

PC(program counter)>>>> EP >>>reset of startup code

What startup code do ???

1-init _stack

2-copy (.data) from flash to RAM

3-reserve (.bss) in RAM

4-branch/jump >> main

case 2: bootloader

developer >>> EP >> bootloader (startup+main)

what boot loader do??

a-load source >>> dist at runtime

b-init modules

c-jump to startup (SW)>>>main

running moods:

a-ROM mode: case1

ram: stack/change data

rom: entry point/exe code +const data

b-RAM mode: case2

rom: entry point/code image

ram: stack/exe code /change da

ROM mood	RAM mood
• Very simple • Fixed code address • Relative small code • Require smaller memory	• Complex • Relocatable mode • Fast why??(RAM > ROM)