Lecture 8

Content

Compile in CMD | Cross Compiler vs Native Compiler | Macros

Other Preprocessor Directives and Operators

Peak to Compilation Process

Preprocessor Conditional Compilation Directives

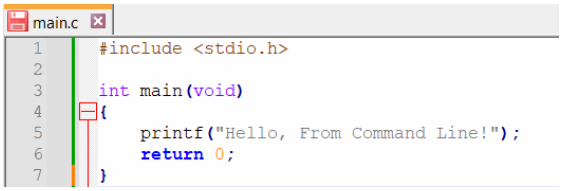
Static vs Dynamic Linking | Useful Command Lines

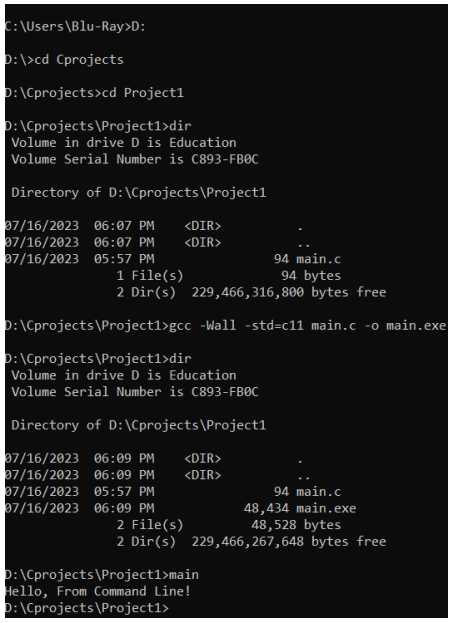
Some Useful Command Lines

- `cd` Command
  - Used to enter a directory if it is followed by a directory name `cd file`
  - Used to exit a directory if it is followed by two dots `cd ..`
- `dir` Command
  - Used to display the content of the current directory
- `cls` Command
  - Used to clear the command prompt
- `type nul > file.extension` command
  - Creates a file in the current directory.
  - We replace `file` with the file name
- Up\Down arrows

- Go back\forward to any previous commands we typed

- To Compile
  - .c file
    - `gcc -Wall -std=c11 main.c`
    - `gcc -Wall -std=c11 main.c -o main.exe` it simply set the output (.exe) file name
    - `gcc -Wall -std=c11 -c main.c -o main.o` it makes the compiler generate an object file
  - .cpp file
    - `g++ -Wall -std=c++14 main.cpp`
    - `g++ -Wall -std=c++14 main.cpp -o main.exe` it simply changes the .exe file name
- To run the .exe file We simply write the file name
- `<Partition name>:` Command
  - It takes you to another partition `D:`
- Example to compile .c file in the command prompt



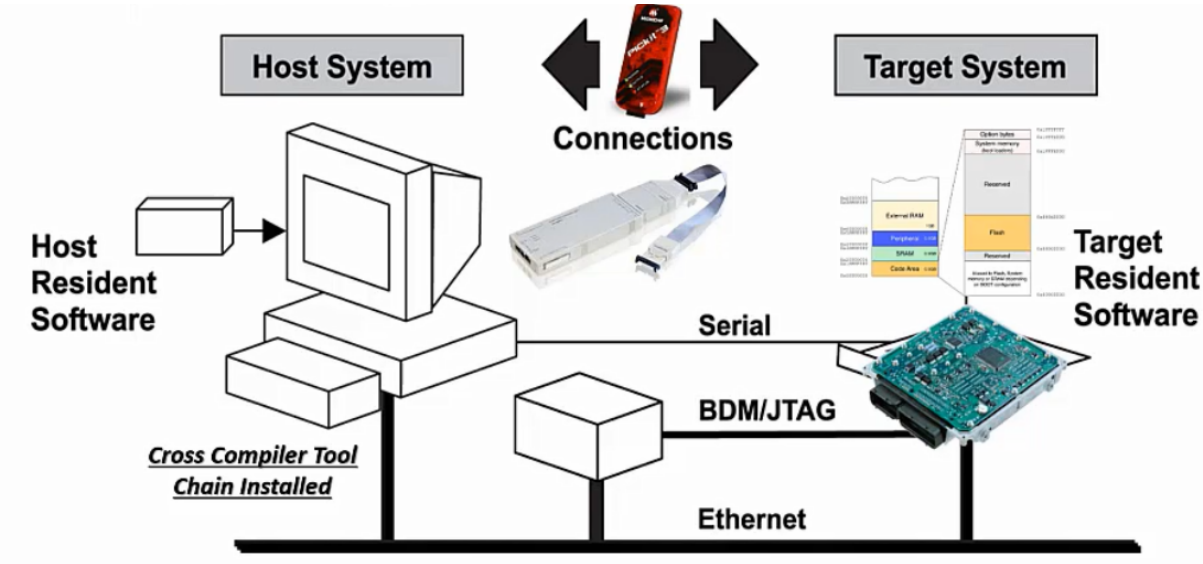


Cross Compiler vs Native Compiler

Native Compiler

- Compiler that generates a code for the same platform on which it runs
- It converts the high level code into computer’s executable format
- The code generation\compilation and running the executable happened on the same platform
- Example: Turbo C and GCC compiler

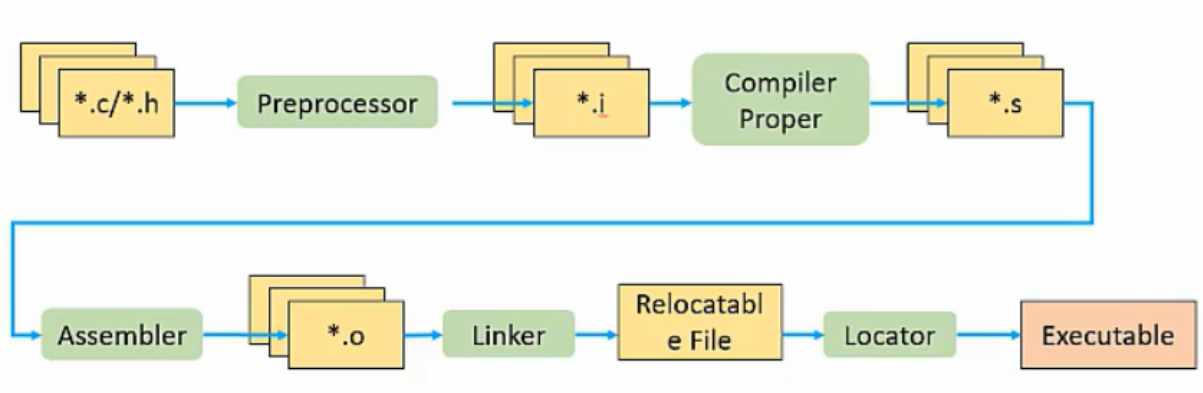
Cross Compiler



- The code will be Written in a platform (Host) and will be executed in another platform (Target)
- Compiler that generates executable code for another platform on which the compiler is running
- Example: GCC compiler for ARM Embedded Processors (GNU Arm Embedded Toolchain)
  - The output executable will run in to ARM based MCU

Peak to Compilation Process

- Tool Chain:
  - It is a set of executable files the compiler use to convert the .c files to .exe file
  - You will find the Tool Chain files in the bin folder in the compiler directory
    - cpp.exe → Preprocessor
    - gcc.exe → Compiler
    - as.exe → Assembler
    - ld.exe → Linker

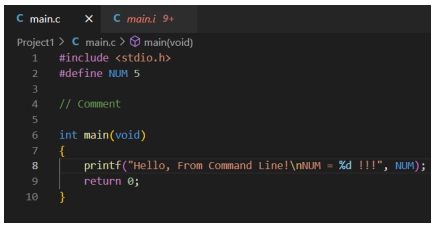


Preprocessor

- It takes a .c file and generates a .i file (preprocessed → postprocessed)
- The preprocessor file name in the Tool Chain is `cpp.exe`
- To call the preprocessor for a .c file we use following command `cpp main.c > main.i` or `cpp -Wall -std=c11 main.c -o main.i`

```
D:\Cprojects\Project1>cpp main.c > main.i

D:\Cprojects\Project1>
```

- It does text replacement. replace each # with its equivalent text, and replace each comment with a single space



- If we have a user-defined library in a folder different from the project folder, we can include it with its path. Example: `#include "C:\Users\Username\OneDrive\Desktop\Cprojects\Project1\Libraries\myLibrary.h"`
- We cannot define a function more than one time. So, if we include the .c file that contains the functions declarations in more than one .c file we will get an error "multi-definition error"

Compilation process Commands

- To make the compiler takes the (.i) file and generate a (.s) file we use following command `gcc -Wall -std=c11 -S inputFile.i -o outputFile.s` or `gcc -S inputFile.i -o outputFile.s`

```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11 -S main.i -o main.s
```

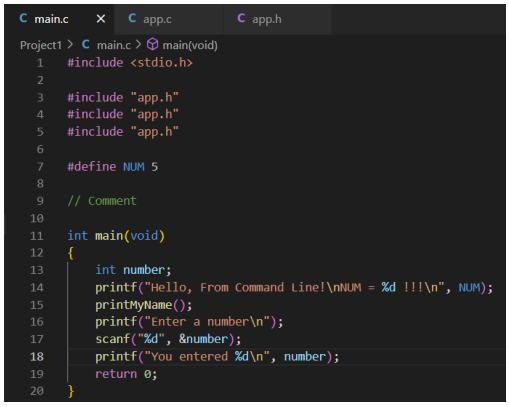
- To make the assembler takes the (.s) file and generate a (.o) file we use following command `as -Wall inputFile.s -o outputFile.o` or `as inputFile.s -o outputFile.o`

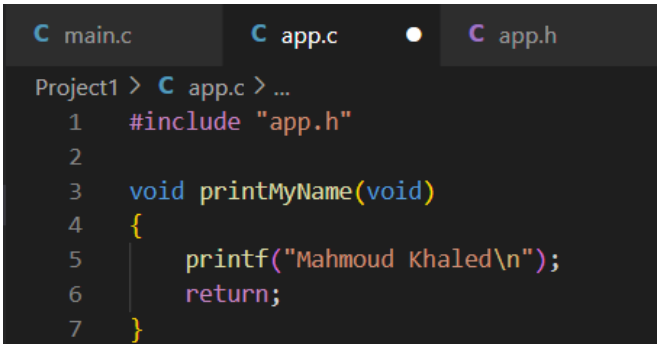
```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>as -Wall main.s -o main.o
```

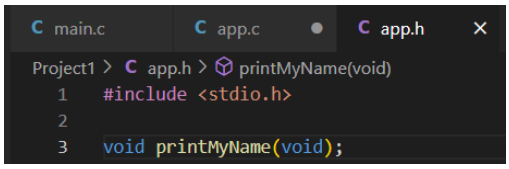
- To Link the (.o) files and generate a (.exe) file we use following command `gcc -Wall -std=c11 main1.o main2.o ... -o main.exe`

```
C:\Users\Blu-Ray\OneDrive\Desktop\Cprojects\Project1>gcc -Wall -std=c11  main.o -o main.exe
```

- A step by step compilation process on command prompt







Static vs Dynamic Linking

- Static and dynamic linking are two different methods of linking object files together to create an executable file in a software development environment.
- Static linking involves linking the libraries used by the executable file directly into the executable itself, creating a standalone executable file that doesn't depend on external libraries at runtime.
- Dynamic linking involves linking the executable file to external libraries at runtime, resulting in a smaller executable file that depends on the libraries being available on the system.
- Static linking can result in larger executable files and may require recompiling the entire program if any of the linked libraries are updated.
- Dynamic linking allows for more efficient use of system resources but requires the libraries to be installed on the system separately and may result in version compatibility issues.
- Choosing between static and dynamic linking depends on the specific requirements of the project, such as the need for portability, performance, and ease of maintenance.

Macros

Macros Best Practices

- Macros Name is written in upper case letters

- If the macro name is a multi-word name we separate between then with `_`
- We cannot write any comments with the same line with the macro except the last line
- In function like macro we wrap each parameter with round brackets
- macro doesn’t end with semi-column

  `#define MACRO_NAME (300.0)  // this is a macro`
- The preprocessor doesn’t replace any text between double quotes. Example:

```
#include <stdio.h>
#define MY_NAME "Mahmoud Khaled Mohamed"

int main(void)
{
    printf("My name is MY_NAME"); // here the preprocessor doesn't replace MY_NAME
    return 0;
}
```

Multi-line Macro

- Syntax for multi-line macro

```
# \
define \
MACRO_NAME \
5000
```

- In multi-line macro if we need to write any comments it must be in the last line

```
# \
define \
SET_BIT(X, BIT_POSITION) \
(X) |= (1 << (BIT_POSITION)) // This function like macro set a bit with value 1 in variable X
```

- Example to multi-line macro. Swapping function using XOR method

```
#define SWAP(X, Y) \
{ \
    *(X) = *(X) ^ *(Y); \
    *(Y) = *(X) ^ *(Y); \
    *(X) = *(X) ^ *(Y); \
}
```

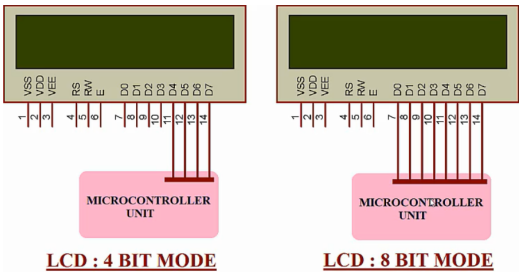
Preprocessor Conditional Compilation Directives

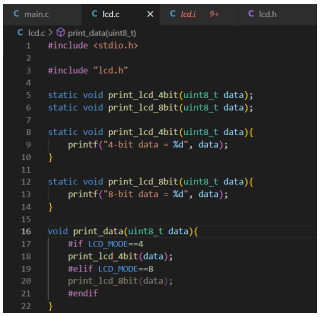
#if (condition) . . . #endif

- It is used to choose between two or more codes so that one of them is compiled under a condition and the other code is neglected
- Syntax

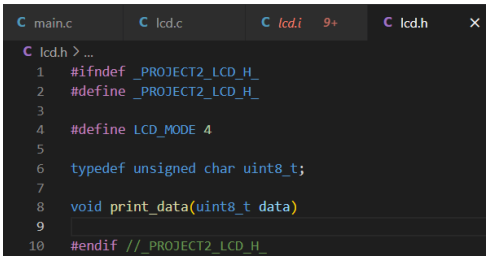
```
#if (/* condition 1 */)
/* Code 1 to be compiled */
#elif (/* condition 2 */)
/* Code 2 to be compiled */
#else (/* condition 3 */)
/* Code 3 to be compiled */
#endif
// the #endif must end the structure of the preprocessor directive
```

- Real world example: If we have an LCD driver written. The LCD drive contains a code for 8-bit LCD and 4-bit LCD. Here, we need to choose one code of them to compile depending on our LCD.

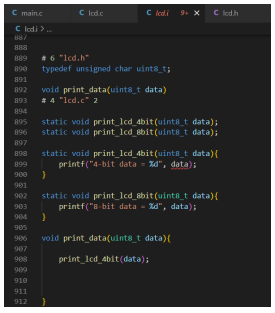




```c
#include <stdio.h>

#include "lcd.h"

static void print_lcd_4bit(uint8_t data);
static void print_lcd_8bit(uint8_t data);

static void print_lcd_4bit(uint8_t data){
    printf("4-bit data = %d", data);
}

static void print_lcd_8bit(uint8_t data){
    printf("8-bit data = %d", data);
}

void print_data(uint8_t data){
    #if LCD_MODE==4
    print_lcd_4bit(data);
    #elif LCD_MODE==8
    print_lcd_8bit(data);
    #endif
}
```



```c
#ifndef _PROJECT2_LCD_H_
#define _PROJECT2_LCD_H_

#define LCD_MODE 4

typedef unsigned char uint8_t;

void print_data(uint8_t data)

#endif // _PROJECT2_LCD_H_
```



```c
# 6 "lcd.h"
typedef unsigned char uint8_t;

void print_data(uint8_t data)
# 4 "lcd.c" 2

static void print_lcd_4bit(uint8_t data);
static void print_lcd_8bit(uint8_t data);

static void print_lcd_4bit(uint8_t data){
    printf("4-bit data = %d", data);
}

static void print_lcd_8bit(uint8_t data){
    printf("8-bit data = %d", data);
}

void print_data(uint8_t data){

    print_lcd_4bit(data);

}
```

- We can eliminate a piece of code if we put it between `#if 0` and `#endif`. It removes the code in the preprocessing time

```c
#include <stdio.h>

#include "lcd.h"

int main()
{
    #if 0
    printf("Mahmoud khaled mohamed\n");
    return 0;
    #endif
}
```

#ifdef (Macro) . . . #endif

- Used to detect if a macro is defined or not
- Syntax

```
#ifdef (/* macro */)
/* Code 1 to be compiled */
#endif
```

File Guard Preprocessor Directive

- It is used to prevent copying the content of the (.h) file multiple times if it is included multiple times
- Syntax

```
// our .h file
#ifndef _FOLDER_FILE_H_
#define _FOLDER_FILE_H_
/* .h file content */
#endif
```

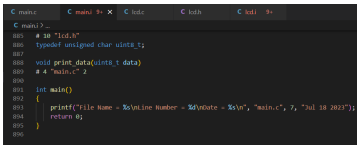
- We can use it to define a not defined macro. Example

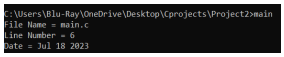
```
#ifndef NULL
#define NULL ((void *)0)
#endif
```

Other Preprocessor Directives

__FILE__, __LINE__, __DATE__ Preprocessors

- `__FILE__`: Replaced with the file name by the preprocessor → string type
- `__LINE__`: Replaced with the line number name by the preprocessor → int type
- `__DATE__`: Replaced with the file date by the preprocessor → string type





#error “Error message”

- Used to make the preprocessor generates an error message
- It stops the compilation process
- Syntax: `#error “Error Message”`
- Example

```
#ifdef TEST
#error "Test is defined"
#else
#error "Test is NOT defined"
#endif
```

#line “New Line Number” “New File Name”

- Used to set a line number instead of the default number
- Syntax: `#line “New Line Number” “New File Name”`
- Example

```
#include <stdio.h>

int main(void)
{
#line 33 "new.c"
  printf("File Name = %s\tLine Number = %d\n", __FILE__, __LINE__);  // File Name = new.c    line Number = 33
  printf("File Name = %s\tLine Number = %d\n", __FILE__, __LINE__);  // File Name = new.c    line Number = 34
  return 0;
}
```

#pragma

- Compiler dependent. #pragma Commands differ from compiler to another. To know pragmas offered by the compiler you have to read the compiler’s documentation
- It may affect the code portability. Because of that it is rarely used
- Most of the compilers define the command `#pragma once` to do the functionality of file guard. But file guard is more used.
- `#pragma GCC poison identifier` this pragma is used specially with GCC compiler. It is throw an error if the identifier specified is used in our file. Example

```
#pragma GCC poison printf
. . .
printf("Hello, World!");  // error
. . .
```

- `#pragma GCC warning “Warning message”` this pragma is used specially with GCC compiler. It is throw an warning at preprocessing time.
- `#pragma GCC error “error message”` this pragma is used specially with GCC compiler. It is throw an error at preprocessing time, and stops the compilation

---

Preprocessor Operators

defined(MACRO)

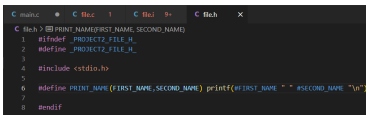
- Return true if the macro passed into it is defined. Otherwise, It returns false
- Syntax

```
#define TEST 2

#if definded(TEST)
/* code */
#endif
```

- We can write logical expression with it like `!defined(test1) && defined(test2)`

Stringize Operator

- Used to put the macro next to it between double quotes
- Syntax `#MACRO`
- Example



Continuation Operator

- It is used to indicate that the macro it continued in the next line
- Syntax

```
#define FUN(PAR_1,PAR_2) \
{ \
    /* code */ \
}
```

- Example

```
#define SWAP(X,Y) \
{ \
    X = X ^ Y; \
    Y = X ^ Y; \
    X = X ^ Y; \
}
```